ANSI C Toolset User Guide

INMOS Limited

INMOS is a member of the SGS–THOMSON Microelectronics Group

72 TDS 345 01

October 1992

INMOS Document Number: 72 TDS 345 01

Contents overview

Contents

Preface

Host versions

The documentation set which accompanies the ANSI C toolset is designed to cover all host versions of the toolset:

- IMS D7314 – IBM PC compatible running MS–DOS
- IMS D4314 – Sun 4 systems running SunOS.
- IMS D6314 – VAX systems running VMS.

About this manual

This manual is the *User Guide* to the ANSI C toolset and is divided into two parts: '*Basics*' and '*Advanced Techniques*' plus appendices. In addition some chapters are generic to other INMOS toolsets.

Differences from the previous release of the ANSI C toolset are listed immediately after this preface.

The basic section introduces the transputer and the toolset; provides an overview of the development cycle and then provides a chapter on each of the following:

- Getting started – a tutorial.
- Parallel programming using a single transputer.
- The configuration process.
- Loading programs onto a transputer network.
- Debugging programs with the toolset debugger `idebug`.

The advanced section is aimed at the more experienced user and covers the following topics:

- Advanced use of the configurer including placing code and data at specific memory locations and the software virtual through-routing mechanism.
- Mixed language programming.
- Developing programs for EPROM.
- Developing code which may be dynamically loaded.

The appendices provided in the *User Guide* include a glossary of terms and a bibliography.

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 345 01 ANSI C Toolset User Guide* (this manual)
- *72 TDS 346 01 ANSI C Toolset Reference Manual*

  Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products i.e. the occam and FORTRAN toolsets and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.

- *72 TDS 347 01 ANSI C Language and Libraries Reference Manual*

  Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details are also provided about how to modify the runtime startup system, although only the very experienced user should attempt this.

- *72 TDS 348 01 ANSI C Optimizing Compiler User Guide*

  Provides reference and user information specific to the ANSI C optimizing compiler. Examples of the type of optimizations available are provided in the appendices. This manual should be read in conjunction with the reference chapter for the standard ANSI C compiler, provided in the *Tools Reference Manual.*

- *72 TDS 354 00 Performance Improvement with the DX314 ANSI C Toolset*

  This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual.* Note: details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide.*

- *72 TDS 355 00 ANSI C Toolset Handbook*

  A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual.*

- *72 TDS 360 00 ANSI C Toolset Master Index*

  A separately bound master index which covers the *User Guide*, *Toolset Reference Manual*, *Language and Libraries Reference Manual*, *Optimizing Compiler User Guide* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.

occam and FORTRAN toolsets

At the time of writing the occam and FORTRAN toolset products referred to in this document set are still under development and specific details relating to them are subject to change. Users should consult the documentation provided with the corresponding toolset product for specific information on that product.

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
`Teletype`	Used to distinguish command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis . . .	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

Differences from previous issue

This section provides a brief list of the differences between this and the previous release of the ANSI C toolset; full details can be found in the relevant section of the toolset documentation.

New and changed features:

Host types:

- The host types supported by this toolset are: IBM PC 386, Sun 4 running SunOS 4.1 or later and VAX under VMS.
- Bootable versions of hosted tools are no longer supplied and as a result the sources of all tools are can be found in the `tools` directory. The `itools` and `iserver` directories found in previous releases, no longer exist. See the accompanying delivery manual for details of source directories.

Compiler

- A new optimizing C compiler is supplied which generates code for any 32–bit transputer. It does not support 16–bit transputers or debugging information. See the *Optimizing Compiler User Guide* for details.
- A compiler command line option is provided for invoking the assembler, see the appendices of the *Toolset Reference Manual.*
- A new compiler option `FC` directs the compiler to treat plain `chars` as `signed chars`, see the *Toolset Reference Manual.*

Compiler pragmas

- The following new compiler pragmas are supported:

Pragma	Supported by:	Description
`IMS_nosideeffects`	Optimizing compiler	Marks a function as side effect free.
`IMS_descriptor`	Both the standard compiler and the optimizing compiler.	Creates a TCOFF descriptor for C functions.

The pragmas are documented in the *Toolset Reference manual* and the *Optimizing Compiler User Guide.*

Configuration

- The configurer supplied with this toolset supports virtual routing by using software processes. If the user specifies the `icconf` 'NV' command line option then the configuration will be similar to that produced by the previous toolset. See chapter 6 of this manual.

- The virtual routing processes have certain implications for debugging. To reproduce the results achieved using the previous toolset, use the configurer command line:

  ```
  icconf -g -nv
  ```

  instead of:

  ```
  icconf -g
  ```

- The configuration language has been extended to include attributes which enable the user to specify actual addresses at which code and data is to be loaded into memory. New attributes have also been added to enable the configurer to selectively use particular routes though the network. See chapter 9 of this manual.

Collecting

- The `RO` and `RA` collector command line options for specifying boot–from–ROM output no longer need to be specified for configured programs. This is because configuration options should be used to specify ROM output. See the *Toolset Reference Manual.*

Debugger – see chapter 8 of this manual.

- The debugger `idebug` may be used with programs which use software virtual routing. A new option has been introduced to display virtual links on a processor.
- `idebug` can now debug boot–from–ROM, run in RAM programs.

Memory map

- A new tool `imap` has been provided which will produce a detailed memory map for a collected program. Intermediate memory maps may also be produced by the compiler, linker and collector tools. See the *Toolset Reference Manual.*

New `iserver`

- A completely new version of `iserver` has been supplied with this toolset. See the *Toolset Reference Manual* for details.

Dynamic code loading

- A set of library functions is provided that enable an application to load and execute a process that has been separately compiled and linked. The loaded process is created as an `.rsc` file. See chapter 12 of this manual.

Bootstrap loaders

- The sources of the bootstrap loaders are supplied. The sources are fully commented so that they can be tailored as required. See the appendices of the *Toolset Reference Manual.*

Parallel process stacks

- The support for parallel process stacks has been extended in this release to allow them to exist anywhere in the transputer address space except nested within an existing parallel process stack.

Libraries

- **Startup linker files:** New versions of the startup linker files are supplied. (See chapter 3 of this manual).

Use	Link file	Entry point
Configured programs using full runtime system	`cstartup.lnk`	`C.ENTRYD`
Configured programs using reduced runtime system	`cstartrd.lnk`	`C.ENTRYD.RC`
Non–configured programs using full runtime system	`cnonconf.lnk`	`C.ENTRY`

Note: that the configured and non–configured cases have been separated. In future toolsets, the non–configured case may not be supported.

The linker files supplied with the previous issue are maintained in a modified format in the present toolset for compatibility purposes; they will be omitted in future releases:

Use	Link file	Entry point
Programs using full runtime system	`startup.lnk`	`C.ENTRY`
Configured programs using the reduced runtime system	`startrd.lnk`	`C.ENTRYD.RC`

Note: that `startup.lnk` should only be used for non–configured programs and that the entry point has changed for the reduced linker file.

It is strongly recommended that all applications are configured and that the new linker files `cstartup.lnk` and `cstartrd.lnk` are used for future development.

In addition the following startup files are supplied which do not specify an entry point. They can be used whenever the main entry point of a program is not one of the standard C entry points. For example certain cases of mixed language programming, or when generating code which will be dynamically loaded.

Linker indirect file	Comment
`clibs.lnk`	Lists the library files required for the full library.
`clibsrd.lnk`	Lists the library files required for the reduced library.

- **Reducing library entry overhead:**

  In order to provide flexibility for the user to tailor the runtime system to a particular application, the source code of the startup routines is provided. Guidance on how to modify the startup system is given in the *ANSI C Language and Libraries Reference Manual.*

- **Channels:**

  The definition of type `Channel` is changed to provide the optimizer with better information. If a program restricts its use of channels to the documented uses of the header files, then the program will continue to work.

- **Library functions** See the *Language and Libraries Reference Manual* for full details.

  - The library functions `exit` and `exit_terminate` have been modified with respect to when they terminate the server. They now terminate the server for configured programs. A new function `exit_noterminate` has been added.
    (Note: should `startup.lnk` be used, in the configured case, then `exit` will not terminate the server. This is compatible with the previous release of the toolset).

  - `ProcInit` – the pointer to the stack space to be used may now point anywhere within the transputer address space except into the stack space of an existing parallel process.

  - The functions `ProcJoin` and `ProcJoinList` have been added. Both functions wait for a list of asynchronous processes to terminate. The first function takes a list of pointers to process structures as a parameter; the second function accepts an array of pointers to process structures.

  - The function `_IMS_HOST_IBM370` has been added to support the new host type IBM 370.

  - `ProcRun ProcRunHigh ProcrunLow ProcPar ProcParList ProcPriPar` give a fatal error message if they detect that an attempt has been made to start a process which is already running.

- The header files **fnload.h** and **hostlink.h** have been added to provide functions to support the dynamic loading of code at run-time.
- New functions have been added to the header file **misc.h**:

  call_without_gsb, get_details_of_free_memory, get_details_of_free_stack_space, halt_processor

- The header file **bootlink.h** has been added for the new function **get_bootlink_channels**.
- The following functions have been added and will be implemented inline, provided the appropriate hardware support exists and the appropriate header file is included in the source:

  ```
  BlockMove BitCnt BitCntSum BitRevNBits
  BitRevWord
  Move2D Move2DNonZero Move2DZero
  CrcByte CrcWord
  DirectChanIn DirectChanInChar DirectChanInInt
  DirectChanOut DirectChanOutChar DirectChanOutInt
  ```

  In addition the functions: **ProcGetPriority ProcReschedule ProcTime** are implemented inline where possible.

- In addition to the two new CRC inline functions, two further cyclic redundancy support functions: **CrcFromLsb** and **CrcFromMsb** have been added.
- The functions **ChanInChar** and **ChanOutChar** have had the type of **char** changed to **unsigned char**.
- The type definition of **clock_t**, a parameter to the function **clock**, has changed from an **unsigned long** to an **unsigned int**. **CLOCKS_PER_SEC** is now sensitive to the priority of the calling process i.e. it is a different value depending on whether the priority of the process is high or low. In addition the two constants **CLOCKS_PER_SEC_HIGH** and **CLOCKS_PER_SEC_LOW** have been added to the header file **process.h**. Care should be taken when using the function **clock** on 16–bit transputers at high priority.
- The library **callc.lib** provides four occam procedures for assisting with mixed language programming. This library is not a C library but is now supplied with this toolset rather than with the INMOS occam 2 toolset.

Documentation

- The toolset documentation has been substantially reworked and restructured. What used to constitute the '*User Manual*' has now been split into two volumes: the '*Toolset User Guide*' and the '*Toolset Reference Manual*'. The '*Toolset Reference Manual*' has been renamed the '*Language and Libraries Reference Manual*'.

- Many of the chapters in the new '*Toolset User Guide*' and the '*Toolset Reference Manual*' are generic to several INMOS Toolset products.

Features removed:

Tools

- The two file formatting tools `icvlink` and `icvemit` have been removed.

Common command line options

- The common command line options '`L`', '`XM`' and '`XO`' to load transputer hosted versions of the tools have been removed.

Libraries

- All 3L concurrency functions have been removed.
- The functions `_memcpy` and `_strcpy` have been removed.
- The feature which allowed the first array of input channels and first array of output channels (found in the configuration interface description for a C program) to be accessed from the `main` function argument list as `in` and `out` is no longer supported.

Compiler pragmas

- The `inline_string_ops` parameter to `IMS_on` and `IMS_off` is no longer supported.

Basics

1 Introduction to transputers

This chapter introduces transputers and the programming models which may be adopted when designing programs for the transputer. It describes the main features of the transputer and transputer systems, and introduces the Communicating Sequential Process (CSP) model of parallel processing.

1.1 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware and external communication links. They can be connected one-to-another by their INMOS serial links in application-specific ways and may be used as building blocks for complex parallel processing networks or as powerful dedicated microprocessors.

The transputer is a complete microcomputer on a single chip. In addition to hardware support for concurrent programming and inter-processor communication it contains:

- A very fast (single cycle) on-chip memory.
- A programmable memory interface that allows external memory and memory mapped devices to be added with the minimum of supporting logic.
- System services for integrating transputer systems.
- Real time clocks
- On the T8 series, an integral floating point unit.

Figure 1.1 shows the generalized architecture of the INMOS family of 32–bit transputers. 16–bit transputers are also available.

1.1.1 Transputer links

Links allow processes running on connected processors to exchange data and synchronize their activity. Support for link communications is implemented in hardware on each transputer chip. Communications down links operate concurrently with the processing unit and data can be transferred simultaneously on all links. Most transputers have four links except the IMS M212 and T400 transputers which have just two links.

Transputer links allow tools such as debugging programs to examine memory directly, from a remote processor. Links also provide a means of loading programs onto a network from the host down a single transputer link. Alternatively a network can be loaded via its links from a ROM on a single transputer.

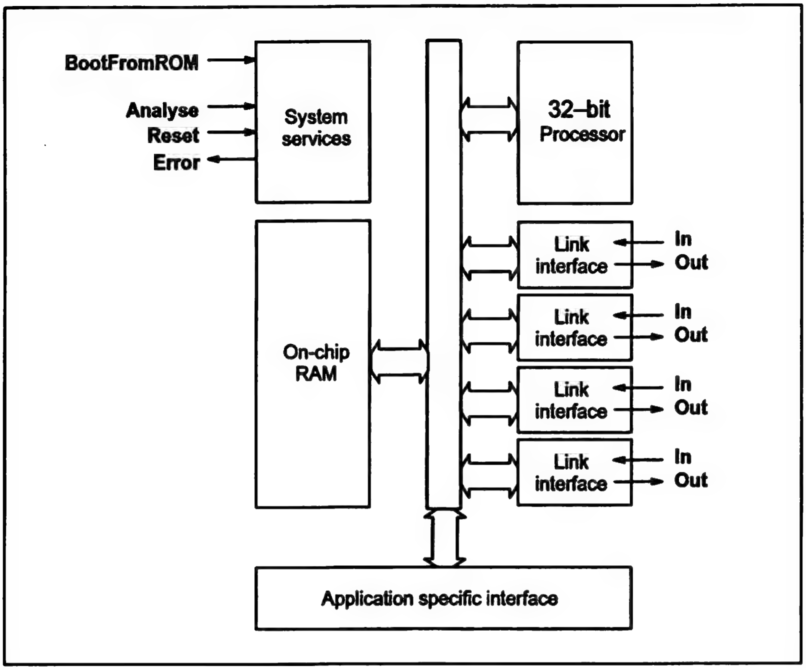


Figure 1.1 Transputer architecture

1.1.2 Process scheduling

Each transputer has a highly efficient run-time scheduler for time-sharing user application processes running on the same transputer. Within a single transputer communication between processes is supported using single words in memory. Processes waiting for input or output, or waiting for a time-slice, consume no CPU resources, and process context switching time is often less than one microsecond.

1.1.3 Real time programming

Features of the transputer provide direct hardware support for real time programming. The key features are:

- Direct and efficient implementation of parallel processes in hardware.
- Prioritization of parallel processes.
- Simple implementation of interrupt handling software.
- Easy programming of software timers, allowing close control of timing and non-busy polling.
- Placement of variables at specific addresses in memory, for accessing memory mapped devices.

Direct support for these features can be found in the current range of INMOS language toolsets, which use a common code format to facilitate code compatibility.

1.1.4 Multitransputer systems

Multitransputer systems can be built very simply using the four high speed links; only two wires are required to connect two links together. The circuitry to drive the each link is on the transputer chip.

Transputers may be connected by their INMOS links in many configurations, depending on the needs of the application. Some possible arrangements of networks of transputers are illustrated in Figure 1.2.

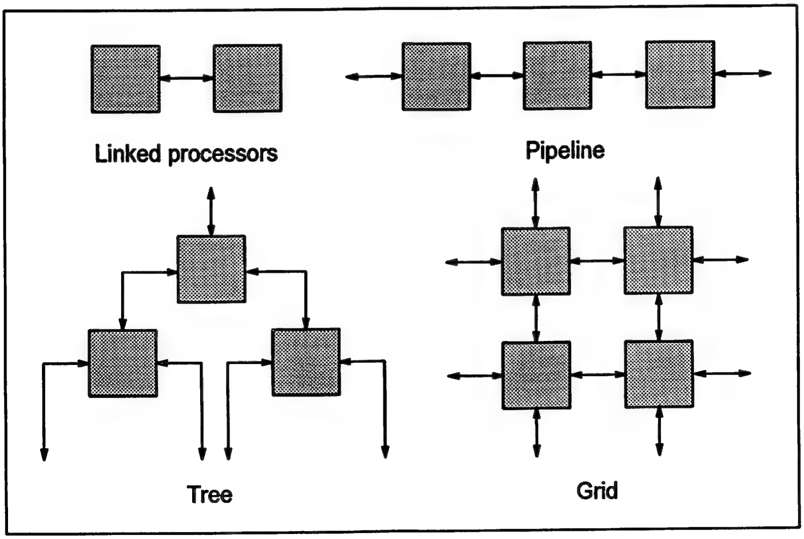


Figure 1.2 Transputer networks

1.2 Programming models

Programs developed for running on a single transputer can be designed using traditional sequential programming methods or they can be designed to exploit parallelism.

Parallelism can be designed into a program at two levels by dividing the program up into a number of independent communicating processes capable of operating in parallel. Such processes can either be run on a single transputer or on a network of transputers. Programs designed for running on a network of transputers must use the parallel processing model. See section 1.2.1.

Sequential programs can be run on a single transputer connected to a host. Such programs can exploit the transputer architecture and software support provided by INMOS toolsets and *i*q systems products, see section 1.3.

1.2.1 Parallel processing model

The abstract programming model which the transputer supports is the Communicating Sequential Process (CSP) model, based on the idea of independent parallel processes communicating through **channels**. Channels are one-way, point-to-point communication paths that allow processes to exchange data and synchronize their activity. (Further details can be found in '*Communicating Sequential Processes*' – C.A.R. Hoare, published by Prentice Hall International).

Each process is built from any number of parallel processes, so that an entire software system can be described in the form of a hierarchy of intercommunicating parallel processes. This model is consistent with many modern software design methods.

Communication between processes is synchronized. When data is passed between two processes the output process does not proceed until the input process is ready and vice versa. Library functions are provided for channel-based input and output.

Communication between software processes running on the same transputer takes place through internal channels implemented as words in memory; communication between processes running on connected processors is driven by the link interfaces and takes place through the transputer links.

1.3 Transputer products

There is a complete family of transputer devices, including: 32–bit and 16–bit processors; a link switch; and an adaptor from a parallel port to a link.

A wide range of INMOS *iq* systems transputer programming boards is available for a range of hosts. These boards can be used for:

- Developing and debugging transputer software
- Improving system performance (as accelerator boards)
- Loading software onto embedded systems
- Building specific transputer networks
- Specific applications such as SCSI interfacing.

1.3.1 Toolset products

The INMOS compiler toolsets are complete cross-development systems for transputers. They allow transputers to be programmed sequentially and in parallel using high-level languages, making optimum use of the transputer's built-in parallel features. The combination of access to parallelism from a high level language and a set of tools for configuring and loading programs on transputer-based systems forms a powerful development system for all parallel and embedded software applications.

2 Overview of the toolset

This chapter introduces the INMOS ANSI C toolset. It briefly describes the features of the compiler, provides an introduction to the runtime library and gives a summary of the tools included in this toolset.

2.1 Introduction

The ANSI C toolset is a software cross-development system for transputers, hosted on 386PC/MS-DOS, Sun 4/SunOS and VAX/VMS systems. It contains a full ANSI C compiler with concurrency support, a multi–language linker, a configurer for mapping programs onto transputer networks, a code collector tool for generating directly loadable files and a combined program loader/host server.

A number of tools are provided to assist with program development: an interactive and post-mortem debugger, librarian and program build tools, an object code lister, and EPROM programming tools. Together, the compiler and its supporting tools form an integrated environment for the development of programs on transputers and transputer–based hardware.

2.1.1 Toolset features

The ANSI C toolset features:

- An ANSI C compiler with concurrency support
- Standard object file format generated by the compiler and linker tools.
- An extensive Runtime Library providing support for concurrent programming based on the communicating process model
- Modifiable runtime system.
- Dynamic code loading facility.
- Support for assembly programming.
- A generic configuration system which facilitates the mapping of software to hardware. The system supports:
    - Mixed language programming.

- Software routing and multiplexing.
- Placement of code and data at specific addresses.

- A comprehensive range of INMOS development tools as listed in table 2.1.

2.1.2 Transputer targets

The ANSI C toolset supports all transputer types in the current range of INMOS transputers. These are listed in appendix B in the *ANSI C Toolset Reference Manual.*

2.2 ANSI C compiler – `icc`

The compiler `icc` is an ANSI standard C compiler with concurrency extensions to support parallel programming for transputers and transputer networks. The ANSI C compiler implementation was developed from the Perihelion Software C Compiler and the Codemist Norcroft C Compiler written by Drs. Arthur Norman and Alan Mycroft.

The ANSI C compiler conforms fully with the X3. 159–1989 ANSI standard for the C programming language. This standard has now been ratified as ISO/IEC 9899:1990 Programming languages – C. The standard specifies the content and defines the interpretation of programs written in C, establishing standards of reliability and maintainability and enhancing portability of programs between systems.

The ANSI standard for C formalizes the original implementation of C as described in *'The C Programming Language'* by Kernighan and Ritchie, and extends it to include a runtime library, some language extensions already in common usage and many other improvements designed to standardize the language.

The original implementation of C will be referred to in the rest of this manual as 'K&R C' and ANSI standard C as 'ANSI C'.

The compiler produces compiled code for specific processor types or transputer *classes* (generic groups of transputers). The compiled object file is in a standard intermediate code format which must be linked, configured and made executable before the program can be run. The executable file consists of code which can be directly loaded onto an initialized network.

Advice about how to create libraries compiled for different processor types is provided in the *ANSI C Toolset Reference Manual* which accompanies this toolset. Appendix B in the *ANSI C Toolset Reference Manual* describes how to compile and link code targeted at a single processor type or at a range of transputers.

Command line options for the compiler are described in the *ANSI C Toolset Reference Manual.* Options are provided to control such facilities as the degree of compiler checking, the suppression of error displays, the suppression of code generation and the output of assembly data to a file.

2.2.1 Concurrent programming

The abstract model used in ANSI C reflects the Communicating Sequential Process (CSP) model of parallel programming. The model maps easily onto the transputer to provide efficient parallel code.

Concurrency is supported within a C main program using a series of predefined data types and a comprehensive set of process handling, channel communication and semaphore manipulation functions.

Software may be broken down into independent linked processes which exchange data and synchronize their activity via channels. Such processes and channels may be mapped onto one or several transputers and are declared within a configuration description.

2.2.2 Standard object file format

The current range of INMOS compilers generate object code in an intermediate form known as *TCOFF* (*T*ransputer *C*ommon *O*bject *F*ile *F*ormat), that can be processed by other tools in the toolset. This standard has been adopted for the development of transputer toolsets and enables modules written in different languages to be freely mixed in the same system.

2.2.3 Preprocessor directives

The ANSI C compiler incorporates an ANSI C preprocessor that allows source file inclusion, conditional and unconditional definitions, and implementation dependent pragmas. The following directives are supported:

```
#include    #else      define
#elif       #undef     #endif
#if         #line      #ifdef
#pragma     #ifndef    #error
```

Details of compiler pragmas are given in the *ANSI C Toolset Reference Manual* which accompanies this toolset.

2.2.4 Include files

Include files can contain declarations, definitions or code. Header files for the runtime library are imported using the **`#include`** directive.

The search paths for files imported with the **`#include`** directive are similar to those for the toolset as a whole (see appendix A of the *ANSI C Toolset Reference*

Manual) but differ in some important respects. Two forms of syntax can be used to specify the filename, one of which allows the search path to be extended by directories specified on the command line. For more details see the *ANSI C Toolset Reference Manual*.

2.2.5 Pragmas

The `#pragma` directive allows some compiler operations to be activated or deactivated in specific sections, of code. Pragmas are defined for setting or overriding compiler options, particularly those concerned with code checking, for defining the size of linker code patches, and for allowing code written in other languages to be called from C.

The pragmas provided with `icc` are listed below:

```
IMS_on            IMS_off            IMS_nolink
IMS_linkage       IMS_modpatchsize   IMS_codepatchsize
IMS_translate     IMS_descriptor
```

Details of pragma syntax and options can be found in the *ANSI C Toolset Reference Manual*.

2.2.6 Error modes

Transputer programs possess an attribute known as the *error mode* which sets the runtime behavior of the transputer. `icc` generates object code in an error mode called UNIVERSAL,which is compatible with error modes generated by other INMOS TCOFF-based compilers.

The other two common modes, which may be encountered in mixed language programs are: HALT which halts the transputer when the program generates a runtime error; and STOP, which stops the errant process but allows the rest of the program to continue. These two error modes are mutually exclusive.

Object modules for a whole program, including those created from different languages, must be in compatible error modes. Error modes for a modular program can be rationalized at link time using the appropriate command line option.

Further information about error modes can be found in the *Toolset reference* manual.

2.2.7 Transputer Program Execution

There are two basic types of programs that can be executed on transputers. One sort can use the full range of runtime library routines and is executed on a transputer network operating as a slave to the host system. The other type uses a reduced subset of the library and can execute on a transputer network without any support from a host system.

If a program requires access to a file system, or other host facilities, then it must operate in the full mode. In order that the host can provide services to the transputer network, there is a program called the `iserver` which executes on the host during the execution of the program on the transputer network.

Further information about the `iserver` can be found in the *ANSI C Toolset Reference Manual.*

2.3 Runtime library

The runtime library is a library of compiled C functions that perform common programming operations. The library contains a complete set of ANSI standard functions plus functions to support the use of the transputer's real time clocks, communications, parallel programming and some non–ANSI extensions.

The concurrency functions are divided into three functional groups: process management, channel communication and semaphore handling. The non–ANSI extensions include a set of input/output (i/o) primitives, a set of short mathematical functions, functions for retrieving information about the host system, and debugging functions.

Libraries are supplied in two forms. The *full* library contains the full set of functions; the *reduced* library contains all functions except those requiring access to the server. The appropriate library is selected by the user by specifying the correct startup file at link time.

Libraries are supplied as object code modules compiled for all transputer types and classes; the correct code is selected by the linker according to the transputer target. Versions are also supplied in different error modes.

Each library module contains either a single function or a few related functions, so that only the minimum code is loaded. The libraries are indexed for quicker reference by the compiler and linker.

2.3.1 Reduced library

The reduced library is available for linking with programs that do not use host system or file i/o or i/o-dependent functions. Examples of these are:

- Code installed in embedded systems
- Code that interacts only with other network processes and has no direct communication with the host.

The reduced library omits all the object code associated with communication with the server, for example, the code that ensures proper close-down of the host server is not loaded. This reduces the size of the library object code that must be linked in with the program. This feature is particularly useful in systems where memory space is limited, such as embedded systems.

The reduced library contains all the functions (including concurrency functions) that are in the full library, but omits those which require the host file server. Channel routines are still included so that modules can still communicate with each other, if not with the host. However, common ANSI functions such as `printf` and `getenv` and i/o dependent functions such as `host_info` are not included,

A few functions from the standard i/o library, not true i/o functions, are available in the reduced library. These are the functions `sprintf`, `sscanf`, and `vsprintf`, which are used to format and de-format strings. These three functions are declared in the header file `stdiored.h`.

2.3.2 Header files

Library functions, like all C functions must be declared before use. Declarations of library functions with associated constants, macros, and definitions are held in a number of library *header files* to ensure that function declarations are of the correct form and that supporting macros and constants are included. Header files are given the suffix `.h`.

The library header files contain groups of routines collected together according to common usage. For example, routines that control standard i/o operations are grouped in the file `stdio.h`. Most header files also contain definitions of constants and macros that are associated with the functions' use.

Many of the header files and function groupings are defined in the ANSI standard. The library extensions which support concurrency and other non–ANSI operations are also grouped for programming convenience; for example, functions for sending data down channels are grouped separately from those which manipulate semaphores.

Some library functions are implemented as macros, and a few are implemented as both functions and macros. The decision about which to use depends on the programming style and personal choice.

2.4 Runtime system

In order to provide flexibility for the user to tailor the runtime system to a particular application, the source code of the startup routines is provided. The source code is written in C and is fully commented so that it can be changed by the user to include only the functions that are actually required. Guidance on how to modify the startup system is given in the *ANSI C Language and Libraries Reference Manual.*

Note: this modifiable runtime system only supports programs which have been configured.

2.5 Dynamic code loading

A set of library functions is provided that enable an application to load and execute a process that has been separately compiled and linked. The loaded process is created as a `.rsc` file, using the collector. Functions are provided that read the `.rsc` file and extract crucial information about the process, such as the size of static required and the location of the entry point. The application can then allocate the space required, load the file and call it.

Similar functions are provided to access a `.rsc` file that has been placed into ROM or RAM, or is provided down a channel.

By adapting the startup code (supplied in source form), the code loaded can be tailored to accept any parameters required.

2.6 Low level programming

The compiler supports low level programming in a number of ways by providing:

- a machine code insertion facility;
- a set of functions which can be compiled inline as transputer instructions;
- a direct user interface to the assembler;
- predefined names which can be used to obtain a limited amount of low level information about compiled code.

2.6.1 Assembly code support

The compiler provides support for inline transputer assembly code in C programs. Sequences of transputer instructions can be embedded in C code using the `__asm` construct.

`__asm` can be useful for implementing low level operations such as controlling peripheral devices, and for optimizing the performance of critical sections of code. It is not intended for the wholesale inclusion of large blocks of assembly code and should not be used for this purpose.

Details of how to use the assembly code insertion facility, with examples illustrating commonly performed operations, can be found in chapter 5 '*Language extensions*' of the '*ANSI C Language and Libraries Reference Manual*'.

In addition a set of functions is provided which enable certain transputer instructions to be compiled inline. The functions include: bit manipulation, block moves, CRC calculation and channel i/o support. A list of the functions can be found in section 1.6.1 of the *ANSI C Toolset Reference Manual*; they are described in full in the '*ANSI C Language and Libraries Reference Manual*'.

2.6.2 Compiler predefines

The two predefined names `_lsb` and `_params` can be used as variables, within an `__asm` construct, to determine the position of a compiled object file's static data and a function's parameter block respectively. For further details see the `icc` chapter in the *ANSI C Toolset Reference Manual.*

2.6.3 Assembly programming

Source code written entirely in assembly language can be assembled by `icc` into TCOFF. The object file may then be mixed freely with other object modules and processed by the toolset.

The assembly language uses a mixture of transputer instructions and assembly directives and is described in appendix C of the *ANSI C Toolset Reference Manual.*

2.7 Configuration system

Transputers can easily be joined together in networks, each transputer operating as a computational node and communicating via its links to neighboring nodes. The simplest possible network is a single transputer, which may be attached to a host, or may be a stand alone system booted from an attached EPROM.

In order to prepare such a network to execute an application, it is necessary to define which piece of the application is to be placed on which node of the network. This process is called configuration and is performed using the configurer tool `icconf`. The user provides a description of the hardware network (the 'hardware network description'), the interconnection of the processes (the 'software network description') and some information on how the software is to be mapped onto the hardware. These descriptions are written in the configuration description file.

The configurer reads the configuration description file and checks that the processes have been compiled for the correct transputer type for the nodes where they are to be placed. It also checks that the configuration can be realized in practice on the hardware described and creates a binary description of the configuration. This is used by the collector tool to produce a file that can initialize the network and execute the application.

Since there are only four physical links (at most) on a transputer, the hardware cannot be configured to connect each node to more than four others directly. However, by adding special software processes the effect can be achieved whereby a process on one node can communicate with other processes on many other nodes of the network, unconstrained by the physical topology of the network. In fact, they do not have to be adjacent nodes. The configurer will add these extra processes automatically, but the user has the ability to control to some degree how this is done. This technique is known as '*software through–routing*'.

2.7.1 Configuration language

The C–like configuration language may be used to configure modules compiled by any INMOS TCOFF compilers. This enables modules written in different programming languages to be combined at configuration time.

The configuration language allows software and hardware networks to be described separately and joined by a software-to-hardware description. The language is a simple declarative language incorporating a full range of high level constructs including replicative and conditional statements.

2.7.2 Software routing and multiplexing

The configurer uses software routing and multiplexing software to implement channel communication over *virtual links*. This allows many *virtual channels* to use a single physical link between processors and enables processes on non–adjacent processors to communicate directly.

Future INMOS transputer devices will implement virtual channel communication directly in hardware. The presence of a software virtual routing configurer in the current toolset provides some of the functionality of future processors and is intended to ease the transition to the next generation of transputer products.

2.7.3 Code and data placement

Normally, the configurer will use up the available memory accessible to a processor by allocating the various parts of the application from the lowest address upwards. However, it is sometimes necessary to specify exactly where a piece of code or data should reside. The configurer allows the user to state where the code, stack, heap and/or static of a C program must be placed in memory.

The transputer has some very fast RAM which the application may be required to use in a special way. The configurer can also be told to avoid this area of memory so that the user has free access to it.

2.8 Mixed language programming

The use of standard TCOFF format allows compiled modules from different language sources e.g. C and occam, to be mixed in the same system, with certain restrictions. This method is described in chapter 10.

Calling modules written in other languages is also possible. C can call occam using a 'nolink' pragma which directs the occam code to be compiled without a static base parameter. (Or a dummy static base parameter can be declared in the occam code). occam can call C by using library routines to set up and terminate the static and heap areas.

In all mixed language calls, parameters and return values passed must be of the correct type. Lists of type equivalents between C, and occam are given in chapter 10.

Where character sets differ between languages, 'translate' pragmas can be used to create acceptable aliases in the calling code.

The compiled modules are then linked and configured in the normal way.

Mixed language programs can be constructed easily using the configuration system. Individual linked units written in different languages can be placed on any transputer in the network; to the configurer all linked units are the same and can be mixed in any combination. The method can also be used for mixing code on the same, or a stand–alone, processor; in this case the processor is simply treated as a single–node network and configured in just the same way.

2.9 Toolset summary

The components of the toolset are summarized in Table 2.1. The individual tools are introduced in chapter 3 which describes the main stages of program development.

Tool	Description
`icc`	The ANSI C compiler. A full ANSI standard compiler with concurrency support. Generates object code for specific transputer targets.
`icconf`	The toolset configurer. Enables the user to define which processes (or main programs) are to be placed on which transputers in a network of transputers. It checks the connections between the processes and adds extra system support processes where necessary.
`icollect`	The toolset code collector. Collects linked units into a single file for loading on a transputer network. Takes as input a configuration data file or a single linked unit.
`idebug`	The toolset network debugger. Supports post-mortem and interactive debugging of transputer programs.
`idump`	The memory dumper. An auxiliary tool for use when debugging programs on the root transputer.
`iemit`	The transputer memory configuration tool. Used for evaluating and defining memory configurations for later incorporation into ROM programs.
`ieprom`	The EPROM program formatter tool. Formats transputer bootable code for input to ROM programmers.
`ilibr`	The toolset librarian. Builds libraries of compiled code.
`ilink`	The toolset linker. Resolves external references and links separately compiled units into a single file.
`ilist`	The binary lister. Disassembles and decodes object code and displays information in a readable form.
`imakef`	The Makefile generator. Generates Makefiles for input to MAKE programs.
`imap`	The map tool which gives the addresses of functions and variables used by the program.
`iserver`	The host file server. Loads programs onto transputer hardware and provides runtime access to host services.
`isim`	The T425 simulator. Simulates program execution on an IMS T425 transputer and provides debugging functions.
`iskip`	The skip loader tool. Used with `iserver` to load programs onto external networks over the root transputer.

Table 2.1 The ANSI C toolset

3 Developing programs for the transputer

This chapter gives an overview of the program development cycle using INMOS toolsets. It briefly describes the purpose of each tool and outlines how to use them in developing, configuring, loading and running transputer programs from the host system. The chapter also provides details of command line defaults, environment variables and outlines some host dependencies.

3.1 Introduction

This toolset is one of a range of cross-development systems designed and developed by INMOS for transputer applications. Toolsets which are available include ANSI C, occam and FORTRAN products.

The toolsets have been designed to make program development as simple as possible. Each toolset features a particular language compiler with full library support and then uses a common set of tools for further development stages. For example, tools are included for: creating libraries, linking code, configuring software to run on transputer networks, producing the program bootable file and for loading the application onto hardware. This means that one development methodology can be used to develop programs using a number of different programming languages. Indeed one of the features of the toolsets is that they facilitate mixed language programming.

The toolset includes support for the following functions:

- building executable code;
- loading and running code;
- debugging programs;
- preparing programs for ROM;
- obtaining information about object files.

3.2 Program development using the toolsets

Programs may be developed on the user's host system before down–loading onto either a single transputer or a network of transputers to run.

Executable code is loaded onto a transputer either from ROM or from the host system via a single transputer link onto the '*root*' transputer i.e. the transputer connected to the host. Loadable code is propagated to any other transputers in the network via the interconnecting transputer links.

Creation of executable code for a transputer or transputer network takes several stages involving the use of specific tools at each stage:

1 **Software design.**

The software designer can specify the components of a system in terms of communicating processes. The overall design can be directly expressed in the parallel constructs of the language.

Alternatively conventional sequential programs can be developed for running on a single transputer.

2 **Write the source.**

Source code can be written using any ASCII editor available on the host system. Code can be divided between any number of source files. Source code must conform to the syntax required by the particular language compiler used. For C this is the ANSI standard; occam source code must conform to the occam 2 language definition and FORTRAN source code to FORTRAN-77 syntax.

3 **Compile the source.**

Each source file is compiled using the appropriate language compiler to produce one or more compiled object files in TCOFF format. Each file must be compiled for the same transputer type or for a transputer class covering several compatible types. (More information about transputer types and classes is given in the appendices of the accompanying *Toolset Reference Manual*). Commonly used object code can be combined into libraries using the librarian `ilibr`.

4 **Link the compiled units.**

The compiled object files and libraries are linked together using `ilink`. This generates a single file called a *linked unit* in which all external references are resolved. The linking operation links in the library modules required by the program, which are selected by transputer type from the compiled library code. Object files for input to the linker can be generated by any TCOFF compatible compiler.

Programs developed for the transputer may comprise one or more linked units, created from separately compiled code and library modules. Linked units are assigned to run on a single transputer or a network of transputers during configuration. A linked unit is the smallest unit of code which may be placed on a transputer.

5 **Configure the program.**

Configuration is the process of defining how the application is to be run on hardware. It is achieved by writing a configuration description, assigning linked units to specific processors and optionally connecting them by channels. By changing the configuration description it is possible to run a program on either a single transputer or on different network topologies. The description is processed by the configurer tool to produce a configuration data file. Configuration is used for both single and multiprocessor transputer programs.

The language used to write the configuration description is determined by the toolset. The C and FORTRAN toolsets provide a common configurer, `icconf` which can be used to configure programs written in C, FORTRAN–77 or occam. Using `icconf`, modules written in different languages can be mixed at configuration level, see chapter 6. The occam toolset configurer `occonf` is designed to exploit the parallel programming model of the language and is specific to the occam toolset.

6 **Generate an executable file.**

Before a program can be run it must be made 'bootable'. This involves adding bootstrap information to make the program loadable and is achieved using the collector tool.

The configuration binary file generated by the configurer is read by the code collector `icollect` which generates a single executable file for a transputer network. The collector can generate either a file which is suitable for booting onto a transputer network via a transputer link or one for booting from ROM. The default behavior of the tool is to produce a boot–from–link executable.

Whether a boot–from–ROM executable is generated is determined by command line options specified to the configurer prior to creating the configuration binary file.

7 **Load and run the program.**

An executable boot–from–link file is loaded and run on the transputer network down a host link using `iserver`. Once loaded the code begins to execute immediately. The server tool maintains the environment that supports the program's communication with the host.

8 **Place in ROM**

Executable boot–from–ROM files for embedded systems, are processed by the `ieprom` tool to produce an output file which is suitable for blowing into ROM. Such files may be configured to run from ROM or from RAM.

Programs to be placed in ROM are often developed first as boot–from–link, until they are error free. They are then prepared for ROM by re–submitting them to the configurer and collector, specifying different command line options, prior to using the eprom tools to format them for ROM.

Program development is supported by additional tools which provide facilities for debugging, creating object code libraries, automating the program build, and obtaining information about object files.

Figure 3.1 summarizes the main development stages.

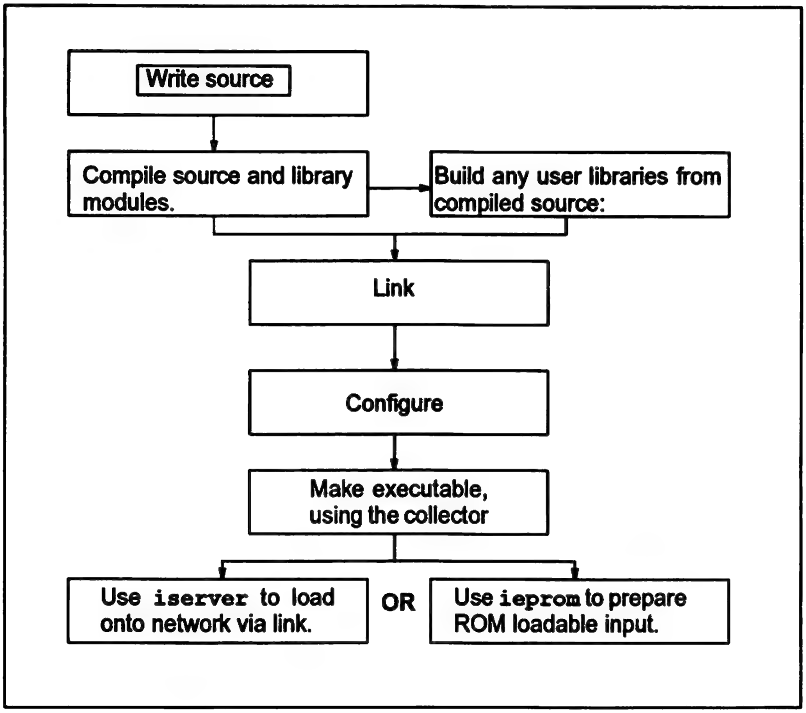

Write source
Compile source and library modules.
Build any user libraries from compiled source:
Link
Configure
Make executable, using the collector
Use iserver to load onto network via link.
OR
Use ieprom to prepare ROM loadable input.

Figure 3.1 Main development stages

3.2.1 Compatibility with previous toolset releases

For single transputer programs the configuration stage of the development process can be omitted. Instead bootable code can be generated directly from the linked unit by specifying a collector command line switch.

This mode of development is not recommended, however, and may not be supported in future toolset releases.

3.3 Compiling

INMOS compilers produce compiled code for specific processor types or for a group of related processors called a transputer class. Each compiler has the same set of options to select the target transputer; these are listed in the appendices to the accompanying *Toolset Reference Manual*. The role of transputer types and classes in compilation and program development is also described in these appendices.

The current range of INMOS compilers generate object code in an intermediate form known as *TCOFF* (*T*ransputer *C*ommon *O*bject *F*ile *F*ormat). This standard has been adopted for the development of transputer compilers and enables modules written in different languages to be freely mixed in the same system.

Supplied with each compiler is a set of language specific libraries which provide runtime support, input/output operations, mathematical functions etc. Support is also provided for language extensions, concurrent programming and software configuration of a network.

The compiler and libraries supplied with this toolset are introduced in chapter 2. Detailed information about the compiler and libraries can be found respectively in the *Toolset Reference Manual* and the *Language and libraries* manual supplied with this toolset.

3.4 Tools for building executable code

Three tools are used in sequence to generate the loadable file from compiled object code:

- `ilink` – the toolset linker which links separately compiled units
- `icconf` (or `occonf` in the occam toolset) – the configurer tool which generates a configuration binary file.
- `icollect` – the code collector which generates a bootable file for a transputer network from the configuration data file.

The configurer works on a configuration source file written by the programmer. The output of the configurer is an information file which is processed by the collector to generate an executable or bootable file. The executable file contains all the information needed to distribute, load, and run the program on a specific network of transputers.

3.4.1 Linker – `ilink`

The toolset linker `ilink` links separately compiled modules and libraries into a single code unit, resolving external references and generating a single *linked unit*. Linked units are referenced directly from configuration descriptions to map software onto specific arrangements of transputers.

Library modules are linked in with the program by the linker startup file (a form of *linker indirect file*) which must be specified on the linker command line. The correct startup file must be specified, depending which version of the compiler or runtime libraries is required, see section 3.11 for further details.

3.4.2 Configurer

The configurer generates configuration information for transputer networks from a textual *configuration description*. The tool prepares the application for configuring on a specific arrangement of transputers by analyzing the configuration description and creating a configuration binary file for the code collector tool to read.

Configuration descriptions are written using the transputer *configuration language* appropriate to the configurer used, see above.

3.4.3 Code collector – `icollect`

The code collector tool `icollect` takes the binary file generated by the configurer (which references the linked code) and generates a single file that can be loaded and run on a transputer network. The collector generates bootstrap and loading code. The output from the collector contains bootable code modules together with distribution information that is used by the loading code to place the correct modules on each processor.

The collector may also generate non–bootable output files which may be dynamically loaded or loaded onto ROM or RAM.

3.5 Loading and running programs

Boot–from–link code for single transputers and transputer networks is output from `icollect` and is loaded onto the transputer hardware using the host file server tool `iserver`. The `iskip` tool can be used in combination with `iserver` to load a program onto an external network, skipping the root transputer (the transputer connected to the host).

Boot-from-ROM code is processed by the eprom programming tools introduced in section 3.7.

3.5.1 Host file server – `iserver`

The host file server `iserver` is a combined host server and program loader tool. When invoked to load a program it both loads the code onto the transputer hardware and provides runtime services on the host for the transputer program such as i/o.

3.5.2 Skip loader – `iskip`

The skip loader `iskip` forces a program to be loaded over the root transputer (the transputer connected to the host). `iskip` is loaded prior to invoking `iserver` for loading user programs onto a transputer board and prevents the root transputer being used as part of the configured network. It continues to run as long as the user program and passes messages between the host and the network.

The tool is useful when debugging programs because it leaves the root transputer free to run the debugger. This avoids the use of `idump` to save the program image and allows the user program to run on a network that would not support the debugger e.g. because it has not enough memory.

3.6 Program development and support

Several tools are provided to assist in program development:

- `idebug` – the interactive network debugger.
- `idump` – the memory dump tool for use with `idebug` when debugging programs on the root transputer.
- `ilibr` – the librarian which generates libraries of compiled code.
- `ilist` – the binary lister which decodes and displays object files.
- `imakef` – the Makefile generator which creates Makefiles for use with MAKE programs.
- `imap` – the map tool which generates a memory map of the functions and variables used by the program.
- `isim` – the T425 simulator tool which enables programs to be executed in the absence of transputer hardware.

3.6.1 Network debugger – `idebug`

The network debugger `idebug` provides post-mortem and interactive debugging for transputer programs. It allows stopped programs to be analyzed from their memory image or from image dump files (*post-mortem* debugging) and supports interactive execution of a program using breakpoints (*breakpoint* debugging). Breakpoints can be set on source lines or memory addresses, variables can be inspected and modified, and the program restarted with new values.

`idebug` provides two debugging environments: a *symbolic* environment which allows a program to be debugged from source code; and the *Monitor page* environment which allows a program to be debugged at machine level.

The debugger inserts no additional code into the program, but uses parallel processing to monitor the program and display its state. This guarantees that the code generated when debugging is disabled will always run in the same way as the final version of the program.

3.6.2 Memory dumper – `idump`

The special debugging tool `idump` is provided to assist with the post-mortem debugging of programs that run on the root transputer. Since `idebug` executes on the root transputer and overwrites the program image, `idump` must be used to save the image to a file which is later read by the debugger.

3.6.3 Librarian – `ilibr`

The librarian `ilibr` creates libraries of compiled code for use in application programs.

A library is a concatenation of compiled files called modules. The linker only links in modules that are required.

Code compiled by compatible TCOFF toolsets can be mixed in the same library.

3.6.4 Binary lister – `ilist`

The binary lister `ilist` decodes object code files and displays data and information from them in a readable form. Command line options select the category and format of data to be displayed.

Examples of the kind of information that can be displayed are symbolic names, code listing, the modular structure and indexing of libraries and external reference data.

3.6.5 Makefile generator – `imakef`

The Makefile generator `imakef` creates Makefiles for specific program compilations. Coupled with a suitable 'make' program it can automate building of executable code and greatly assist with code management and version control. **Note:** a make program is not supplied.

`imakef` constructs a dependency graph for a given object file and generates a Makefile in standard format. In order to make use of the tool a special set of file extensions for source and object files *must* be used throughout program development. `imakef` uses these file extensions to deduce target transputer types and other options. These extensions are described for `imakef` in the *Toolset Reference Manual.*

3.6.6 Memory map tool – `imap`

The memory map tool `imap` takes the text output from the toolset compiler, linker and collector and creates a map of the absolute addresses of the static variables for functions. The memory map is output on the display screen or redirected to a file as the user wishes.

3.6.7 T425 simulator – `isim`

The T425 simulator tool `isim` simulates the operation of the T425 transputer, enabling programs to be executed in the absence of transputer hardware.

Run in interactive mode it provides low level debugging features such as the inspection of variables, registers and queues, disassembly of memory, break points, and single step execution.

Batch mode operation of the simulator allows programs to be executed without entering the debugging environment.

3.7 EPROM programming

Two tools assist with the installation of programs into ROM, namely, the EPROM programmer `ieprom` and the memory configurer `iemit`.

3.7.1 EPROM programmer – `ieprom`

The EPROM programmer `ieprom` converts ROM-bootable files generated by `icollect` into a format suitable for input to ROM programmers. Files can be generated for input to ROM loading programs provided for specific ROMs, or dumped in straight hexadecimal or binary for input to the users' own ROM loaders.

`iemit` output can also be interpreted and the appropriate bit pattern included in the ROM, see below.

3.7.2 Memory configurer – `iemit`

Some transputers have programmable memory interfaces which may be configured for a particular memory design.

The memory interface configurer `iemit` allows specific transputer memory configurations to be evaluated and can output a configuration file for incorporation into ROM by `ieprom`. The completed configuration file can be read by `ieprom` and interpreted for inclusion in the ROM at the correct address. The transputer can automatically read this data when it is reset and use it to configure its memory interface.

3.8 File types and extensions

The current range of INMOS toolsets use, by default, a standard set of file extensions to identify specific files such as source, compiled object, linked units and bootable files. Certain file extensions are assumed by the tools on input, and others generated by the tools on output, unless extensions are explicitly given on the command line. For example the compiler adds the extension `.tco` to the output file unless otherwise specified.

The adoption of a standard system allows file extensions to be omitted on the command line, and permits host file system utilities to be used. The system is designed to form an integrated whole and reflects the architecture of toolset compilation.

The standard set of file extensions is not mandatory and may be modified according to personal choice, unless `imakef` is to be used to build the makefile. `imakef` uses a special scheme to identify processor types and error modes, as described below.

The standard system has the advantage of ready defaults but may not be readily mapped onto existing development schemes. However, if it is decided to adopt a personalized scheme then it should be reasonably formal and controlled, which is especially important across development teams.

Some extensions recognized by the toolset are used for convention only and are not interpreted by the tools in any special way. For example, the `.lib` suffix for library files and the `.inc` suffix for include files are toolset programming conventions.

The main file extensions used in developing transputer programs are listed in Table 3.1. A full list of all file extensions used by the toolset with descriptions of the file types is given in the appendices to the accompanying *Toolset Reference Manual*.

Figure 3.2 illustrates the program development process in terms of the file extension defaults used by the toolsets. The extensions assumed on input and generated on output are used to represent source and target files. Figure 3.2 highlights the differences between the different language toolsets and shows how software can be developed to be loaded onto transputer hardware directly via a transputer link or held in ROM.

Extension	Description
`.btl`	Bootable code file. Created by `icollect`.
`.btr`	Executable code minus bootstrap information. Used for input to the EPROM tool. Created by `icollect`.
`.c`	C source files. Assumed by `icc`, the ANSI C compiler.
`.cfb`	Configuration data (binary) file. Created by the configurer.
`.cfs`	Configuration description (source) file, read by the C configurer `icconf`.
`.f77`	FORTRAN source programs. Assumed by `if77`, the FORTRAN–77 compiler.
`.h`	Header files for use in C source code.
`.inc`	Include files named in `#INCLUDE` compiler directives for occam, or `#include` statements in configuration descriptions or in FORTRAN–77 statements.
`.lku`	Linked unit. Created by `ilink`.
`.lbb`	Library build file. Input to `ilibr`.
`.lib`	Library object file. Created by `ilibr`.
`.liu`	Library usage files. Created and used by `imakef`.
`.lnk`	Linker indirect file. Input to `ilink`.
`.occ`	occam source files. Assumed by `oc`, the occam 2 compiler.
`.pgm`	Configuration description (source) file, read by the occam configurer `occonf`.
`.rsc`	Dynamically loadable code file. Created by `icollect`.
`.tco`	Compiled code file. Created by all INMOS TCOFF compilers.

Table 3.1 Toolset main file extensions

File extensions required by `imakef`

The Makefile generator `imakef` requires a special set of file extensions to be used for compiled and linked object files. The extensions define the architecture of toolset compilation so that `imakef` can trace file dependencies and create the correct sequence of build commands. They are also used to deduce the transputer type and error mode for each unit.

For details of the file extensions that you must use with the `imakef` tool see the appendices of the *Toolset Reference Manual.*

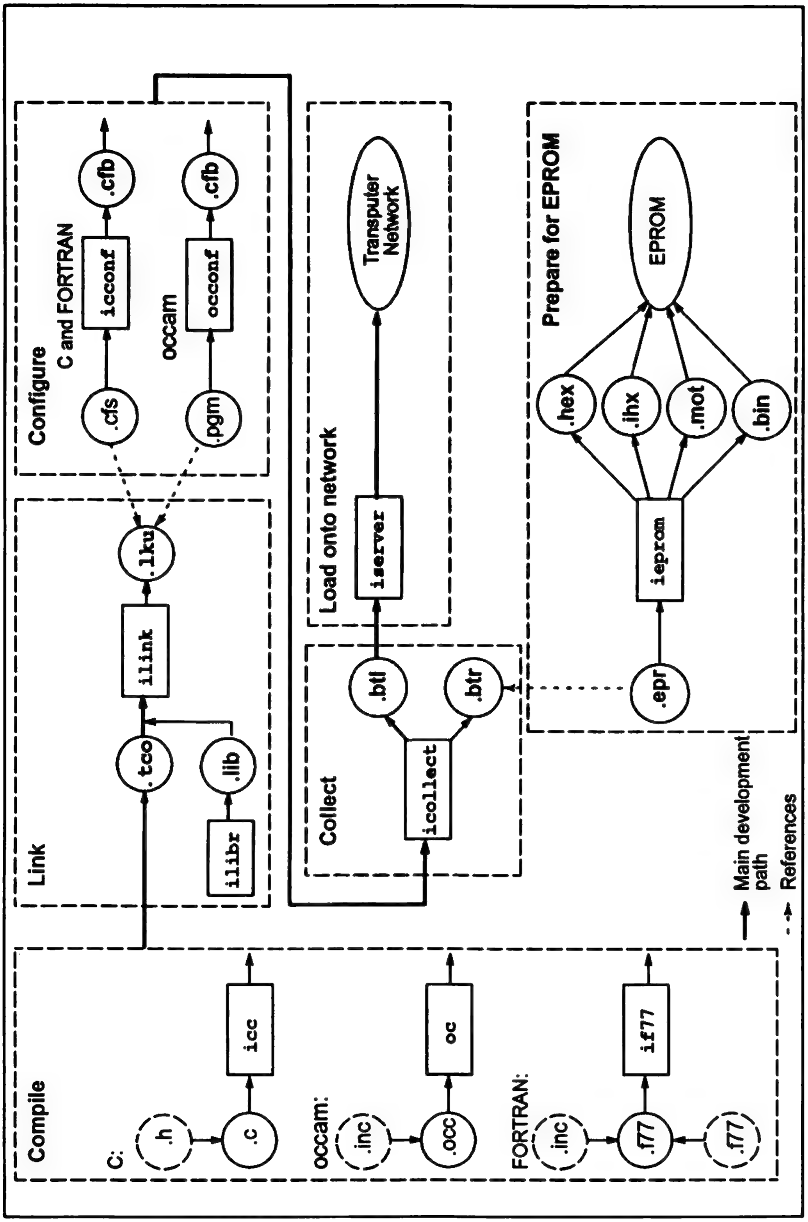


Figure 3.2 Development cycle

3.9 Error reporting

If a tool detects an error in its input, it is reported in a standard format. This contains the name of the tool, a severity level, and some explanatory text explaining why the error occurred. Errors found in files or the file system may also generate a filename and line number. Standardization of the format is designed to improve error reporting and to support automated error handling by host system utilities.

For example:

```
Serious-ilibr-mymod.txt-bad format: not a TCOFF file
```

where: `mymod.txt` is the name of the input file causing the problem.

Note: Messages that are part of the normal operation of the tool, for example, diagnostic messages generated by the compiler, and messages from the debugger and simulator tools, are not required to conform to the standard and may be displayed in special formats appropriate to the tool. The formats will become familiar with use of the tool.

Details of the standard format can be found in the appendices of the accompanying *Toolset Reference Manual*.

3.10 Host dependencies

The toolset uses a host to develop code which is then down loaded onto a transputer or transputer network.

The toolset can be hosted on one of several different platforms, and the tools are designed to blend in as far as possible with the operating system. Source and object code is portable between all systems.

The toolset is available for the following host systems:

- IBM 386 PC (and compatibles) running MS-DOS
- Sun 4 running SunOS
- VAX running VMS.

Differences between the operation of the tools on the various platforms are minor and reflect the 'flavor' of the particular operating system.

Host system dependencies are as far as possible made invisible to the user. The few differences are some minor variations in command line syntax, host-specific library routines, directory names, and environment settings such as search paths and global variables. Each is described briefly below.

Command line syntax

The major difference between host implementations is the use of the host system option prefix. For Unix based toolsets (Sun 4) the prefix character is the dash '–';

for MS-DOS and VAX/VMS based toolsets the prefix character is the forward slash '/'.

For consistency between implementations, the case of options is not significant.

Other command line syntax conventions are identical in all implementations and are described in the appendices of the accompanying *Toolset Reference Manual*.

3.10.1 Filenames

Filenames, with or without a directory path, conform to the normal host system conventions *except* that characters which can be interpreted as directory separators (on any of the supported hosts) must not be used in filenames. This prohibits the use of the following characters: colon ':', semi–colon ';', forward slash '/', backslash '\' ('¥' for Japanese systems), square brackets '[]', round brackets '()', angle brackets '<>', exclamation mark '!',or the equals sign '='.

In addition the linker cannot handle filenames which begin with a hash '#' or with two dashes '--'. These are used to identify commands and comments within linker indirect files.

Where the host operating system allows logical names to be used in place of filenames, such as with VMS, the toolset allows logical names to be used, but the name must be followed by a dot '.'. This prevents the tool from adding an extension, which would generate a host file system error.

3.10.2 Search path

All tools which use or generate filenames use a standard mechanism for locating files on the host system. The same mechanism is used in all operating system versions of the toolset. Briefly, the search mechanism is based on a list of directories to be searched in sequence.

If a directory path is specified only this directory is searched. If the file is not found on the path an error is generated. Relative pathnames are treated as relative to the current directory, i.e. the directory from which the tool is invoked.

If no directory path is specified the current directory is searched followed by the directories specified in the `ISEARCH` environment variable.

Details of how to set up a search path on your system can be found in the Delivery Manual that accompanies the release.

Full details of the mechanism used in file searching can be found in the appendices of the accompanying *Tools Reference Manual*.

3.10.3 Environment variables

The toolsets use a number of environment variables on the host system. Use of these variables is optional but if defined they will influence the behavior of certain of the tools on your system. Further information is given in the *Tools Reference Manual*.

Variable	Meaning
`ICONDB`	Defines the connection database to be used by `iserver`.
`ISESSION`	Defines the session manager configuration file to be used by `iserver`. Defaults to `session.cfg` if not defined.
`ISEARCH`	The search path; i.e. the list of directories that will be searched if a pathname is not specified. Pathnames must be terminated by the standard directory separator character for the system. Used by all tools that read and write files.
`ISIMBATCH`	Used by `isim` to enable/disable batch mode. Values can be `VERIFY` or `NOVERIFY`.
`ITERM`	The file that defines terminal keyboard and screen control codes. Used by `idebug`, `isim` and `iemit`.
`IBOARDSIZE`	The size (in bytes) of memory on the transputer board. Used when loading non–configured programs.
`TRANSPUTER`	Defines the capability (user link name) to be used by the server. Can be overridden by `iserver` command line option.
`IDEBUGSIZE`	The size (in bytes) of memory connected to the root transputer. Used by `idebug`.
toolname`ARG`	Default command line arguments. Applies to certain tools only. See section 3.10.4.

Table 3.2 Toolset environment variables

The exact commands used to define environment variables depend on the operating system. For example, under MS–DOS they are defined using the `set` command; on VAX systems running VMS they can be set up either as logical names or as VMS symbols. Examples of how to set up environment variables can be found in the Delivery Manual that accompanies the release.

For `IBOARDSIZE` and `IDEBUGSIZE` the value can be given in decimal or hexadecimal format. Hexadecimal numbers must be preceded by '`#`' or '`$`'. Leading and trailing spaces may not be given.

Note: If `IBOARDSIZE` is specified incorrectly, for example as a character or string, the system defaults to a board size of 0 (zero) and the program cannot be run. If `IBOARDSIZE` is explicitly set to a very small value a similar error may occur.

3.10.4 Default command line arguments

An environment variable can be defined on the system to specify a default set of command line arguments for certain tools. The variable name must be defined in upper case and is constructed from the tool name by appending the letters '`ARG`'. For example, the variable for `ilink` is `ILINKARG`.

Tools for which a default command line can be defined, and the variables used to define them, are listed below.

Tool	Variable
`icc`	`ICCARG`
`if77`	`IF77ARG`
`ilink`	`ILINKARG`
`icconf`	`ICCONFARG`
`icollect`	`ICOLLECTARG`
`ilibr`	`ILIBRARG`
`ilist`	`ILISTARG`

Table 3.3 Environment variables for invoking tools

Command line parameters must be specified within each variable using the specific syntax required by each tool.

3.11 Linker startup and indirect files

Linker indirect files are text files containing lists of input files and commands to the linker.

A number of linker indirect files are supplied with each toolset. The purpose of these files is to reference various runtime libraries (or in the case of occam, compiler libraries) required to link application programs. When specifying the program modules to be linked, the appropriate linker indirect file must be included on the linker command line, as described in the reference chapter for `ilink` in the accompanying *Toolset Reference Manual.*

3.11.1 ANSI C Toolset

For C the linker indirect files are known as 'linker startup' files. They reference runtime library files which provide the runtime environment for the program and define which version of the C runtime initialization code is used by specifying a main entry point. This is the name of the routine which is called by the transputer bootstrap code or configuration system code, in order to start the C program executing.

Most C programs will require one of the three linker startup files listed in table 3.4. Two files are provided for use with configured programs; one with the full runtime library and one with the reduced runtime library. The reduced library does not support host I/O. (The runtime library is introduced in section 2.3 and described in detail in the *ANSI C Language and Libraries Reference Manual*). It is recommended that all programs are configured.

The third file is provided for use with non-configured programs using the full runtime library.

Special linker startup files which do not specify a main entry point are described in section 3.11.4 below.

Startup file to support:		
Configured programs		Non–configured programs
Full runtime library	Reduced runtime library	Full runtime library
`cstartup.lnk`	`cstartrd.lnk`	`cnonconf.lnk`

Table 3.4 C startup files

`cstartup.lnk`

This linker startup file is used to create linked units which use the full C runtime library and are to be configured using `icconf`. It also specifies a main entry point of `C.ENTRYD`. This is the main entry point of the standard C startup code for configured systems using the full runtime library. `C.ENTRYD` is the first of a sequence of routines which are responsible for setting up the full version of the C runtime system and eventually calling the `main` function. The source of this startup code is supplied with this toolset and is described in section 3 of the *ANSI C Language and Libraries Reference Manual.*

`cstartup.lnk` includes `clibs.lnk` (see 3.11.4). `cstartup.lnk` should only be used if the configurer is also used. The effect of using this linker startup file to create a linked unit which is then passed directly to `icollect`, without using the configurer first, is undefined. It should only be used when the C linked unit created is to have access to host link channels. The startup code assumes that a server exists and will attempt to communicate with it. Thus the effect of its use in an environment where there is no access to the server is undefined.

`cstartrd.lnk`

This linker indirect file is used to create linked units which use the reduced C runtime library and are to be configured using `icconf`. It also specifies a main entry point of `C.ENTRYD.RC`. This is the main entry point of the standard C startup code for configured systems using the reduced runtime library. `C.ENTRYD.RC` is the first of a sequence of routines which are responsible for setting up the reduced version of the C runtime system and eventually calling the `main` function. The source of this startup code is supplied with this toolset and is described in section 3 of the *ANSI C Language and Libraries Reference Manual.*

`cstartrd.lnk` includes `clibsrd.lnk` (see 3.11.4). `cstartrd.lnk` should only be used if the configurer is also used. The effect of using this linker indirect file to create a linked unit which is then passed directly to `icollect`, without using the configurer first, is undefined. It should be used in situations where the C linked unit created has or requires no access to the server. No host link channels are defined.

`cnonconf.lnk`

This linker indirect file is used to create linked units which use the full C runtime library and are suitable for passing directly to `icollect` thereby omitting the configuration stage. Note: this method of program development is only applicable to single processor programs and is not recommended for any new program development as it may be unsupported in future toolsets.

`cnonconf.lnk` specifies a main entry point of `C.ENTRY`. This is a special version of the C startup code which can derive for itself information which is normally supplied by the configurer (as such it is less efficient than the equivalent version of the startup code for configured systems and so use of the configurer is recommended).

`C.ENTRY` is the first of a sequence of routines which are responsible for setting up the full version of the C runtime system and eventually calling the `main` function. `cnonconf.lnk` includes `clibs.lnk` (see 3.11.4). `cnonconf.lnk` should only be used if the configuration stage is to be omitted. The effect of using this linker indirect file to create a linked unit which is subsequently passed to the configurer is undefined. It should only be used when the C linked unit created is to have access to host link channels. The startup code assumes that a server exists and will attempt to communicate with it. Thus the effect of its use in an environment where there is no access to the server is undefined. Indeed, omission of the configuration stage is only possible if the full library is used, therefore there is no equivalent reduced version of this linker indirect file.

3.11.2 occam 2 Toolset

For occam, one of three linker indirect files should be selected according to the target transputer type(s) used, see table 3.5.

Linker indirect file	Target transputers
`occam2.lnk`	T212/T222/T225/M212
`occama.lnk`	T400/T414/T425/T426/TA/TB
`occam8.lnk`	T800/T801/T805

Table 3.5 occam linker indirect files

Each file contains a list of occam library files which may be required to be linked, but which are additional to those explicitly referenced by the program. These include compiler libraries and support for interactive debugging. Depending on the other inputs and options specified on the command line the linker will select which libraries it requires from the supplied indirect file.

3.11.3 Mixed language programs

Mixed language programs require an appropriate linker indirect file for each language used.

For occam, one of the indirect files listed in table 3.5 is always used and when the main program is written in C, one of the files listed in table 3.4 should be used. However, if a non–C program calls in C modules, the standard C startup files are not suitable because they define a C main entry point which would conflict with the actual main entry point of the program. In this case one of the linker files described in section 3.11.4 should be used. These linker files should also be used when incorporating a C program into an occam program as if it were an occam process.

Further information about mixed language programming is given in chapter 10.

3.11.4 Other startup files supplied with the ANSI C Toolset

Two additional linker indirect files are supplied with the ANSI C Toolset:

Linker indirect file	Comment
`clibs.lnk`	Lists the library files required for the full library.
`clibsrd.lnk`	Lists the library files required for the reduced library.

Table 3.6 C linker indirect files referencing libraries only

Unlike the files listed in table 3.4, `clibs.lnk` and `clibsrd.lnk` do not specify a main entry point. They can be used whenever the main entry point of the program is not one of the standard C entry points, for example certain mixed language programs and when producing code which will be dynamically loaded.

`clibs.lnk` should only be used when the C linked unit created is to have access to host link channels. The effect of using in an environment where there is no access to the server is undefined.

`clibsrd.lnk` should be used in situations where the C linked unit created has or requires no access to the server. No host link channels are defined.

3.12 Unsupported options

A number of tools have various command line options beginning with '`z`'. These options are used by INMOS for development purposes and have not been designed for users. As such they are unsupported and should not be used. INMOS cannot guarantee the results obtained from such options nor their continued presence in future toolset releases.

4 Getting started

This chapter outlines how to compile, link and prepare a program for execution on a transputer, using the sample programs in the subdirectory **`simple`** of the toolset **`examples`** directory.

4.1 Outline procedure

In order to create an executable program, a number of things must be done:

1 The source files are compiled with the ANSI C compiler. The compiler creates from each source file an object file.

2 The object files are linked together along with any libraries required to create a file known as a linked unit. Each linked unit contains the code and data necessary to execute as a main program.

3 The linked units are then configured onto a transputer network. In the case of a single program on a single transputer, there is a short cut available here. However, it is strongly recommended that development is made by using the full facilities of the configurer. There are many advantages to this which will become apparent as the procedures are described.

4 The program can then be loaded and run from the host by using the host file server, **`iserver`**. A bootable program contains everything necessary for execution on the transputer network and it will start automatically after it has been loaded.

4.2 Running the examples

In the following examples, the programs are compiled and executed on a single T425. If you have some other transputer, then you should make the necessary changes to the command lines and configuration file as indicated.

The examples assume that the environment variable **`TRANSPUTER`** has been defined to specify the name of a User Link to use for accessing the transputer network, and that a connection database file exists to define that User Link. See the delivery manual which accompany this toolset and the **`iserver`** documentation (chapter 13 of the *ANSI C Toolset Reference Manual*) for more detail.

(Command line options for specifying other transputer types, are listed in appendix B of the *ANSI C Toolset Reference Manual).*

4.2.1 Sources

The sources of all the examples are held in the directory **`examples/simple`**.

4.2.2 Example command lines

In the examples below, the command lines are written in both the UNIX form with the option character `'-'` , and in non–UNIX form with the option character `'/'` for MS-DOS and VAX/VMS systems. Choose the one that is applicable to you.

4.2.3 Using the simulator

If there is no transputer available, then you can use the simulator `isim` to run the bootable program, provided it is built for a single T425.

4.3 A simple sequential program

The following procedure shows how to build and run a simple "Hello World" program. The source for this program is in the file `hello.c`.

4.3.1 Compiling

To compile the program use the command line:

```
icc hello -t425
```
for UNIX systems

or:

```
icc hello /t425
```
for MS-DOS or VAX/VMS

The compiler assumes an extension of `.c` if none is given.

If you have a different transputer, then supply the corresponding transputer type (e.g. `t800`) instead of `t425`. This is necessary for the compiler to generate the correct code for the transputer that will execute it.

The object file that is produced will have the name `hello.tco`.

4.3.2 Linking

The compiled code must now be linked with the C runtime library. To do this use the command line:

```
ilink hello.tco -t425 -f cstartup.lnk
```
(UNIX)

or:

```
ilink hello.tco /t425 /f cstartup.lnk
```
(MS-DOS and VAX/VMS)

The file `cstartup.lnk` is known as a *linker indirect file* and contains the names of the library files that must be searched for the functions that the program calls. The linker will select from the library the correct ones for the transputer target.

It is always good practice to specify to the linker what the transputer target is, since it is possible to produce code that can execute on a range of transputers and the linker must then be told which the actual target will be.

The linker creates a linked unit in the file `hello.lku`.

4.3.3 Configuring

In order to configure the program, a description is required of the network it is run on. The file `hello.cfs` contains such a description.

You should look at this file and edit it if it does not correspond to the hardware you actually have.

The file `hello.cfs` contains the following:

```
/* (c) Copyright INMOS Ltd 1992. All Rights Reserved.*/

/* The configuration uses one processor, connected to the host, */
/* on which is placed a lone main program */

T425 (memory = 1M) Single; /* A T425 with at least 1Mb of memory */

connect Single.link[0] to host; /* Connected to the host on link 0 */

/* Describe the program */
/* The interface of a normal C program is that the C file system */
/* needs access to the host file system. The host runs the iserver */
/* and is connected to the transputer network by an edge called */
/* 'host'. */

process (stacksize = 4K, /* 4Kb of stack used */
        heapsize = 50K, /* 50 Kb of heap requested */
        interface (input FromHost, output ToHost)
        ) Simple;

input HostInput; /* Define the edge from the host */
output HostOutput; /* Define the edge to the host */

connect Simple.FromHost to HostInput; /* connect the process to the */
connect Simple.ToHost to HostOutput; /* software edges */

/* Mapping description */

use "hello.lku" for Simple; /* name of the linked unit to use */
place Simple on Single; /* put the process on the processor */

place HostInput on host; /* say that the software edges are */
place HostOutput on host; /* mapped onto the hardware host edge */
```

In this file, there is one line:

```
use "hello.lku" for Simple; /* name of the linked unit to use */
```

which includes the name of the file containing the linked unit.

In order to configure the application for the network, the configurer is invoked as follows:

```
icconf hello.cfs
```

and it produces a file called `hello.cfb` which contains all the information about where the different parts of the program are to be placed.

4.3.4 Collecting

The next stage is to collect all the parts of the program and combine them into a file which can be loaded onto the transputer for execution. This file is known as a bootable file. The collector is invoked by:

```
icollect hello.cfb
```

The result is a bootable file called `hello.btl`.

4.3.5 Loading and Execution

In order to load and execute the program, it must be placed in the transputer's memory and started. This is done automatically by the `iserver`, a tool which resides on the host, but loads a file onto a network, and then listens for any requests the application may make for host services. The server is invoked by the command line:

```
iserver -sb hello.btl
```
(UNIX)

```
iserver /sb hello.btl
```
(MS-DOS and VAX/VMS)

The greeting:

```
Hello World
```

should then be displayed.

If the program has been built for a single T425 transputer then it may be executed on the simulator instead of the transputer. The simulator is invoked by the command line:

```
isim -bq hello.btl
```
(UNIX)

```
isim /bq hello.btl
```
(MS-DOS and VAX/VMS)

4.3.6 A short cut

When the application is fully debugged, then an alternative method can be used to create a bootable file from a linked unit. This method can only be used if the application consists of a single process running on a single processor and is to be booted from a transputer link attached to a host. It is not applicable to stand alone systems, nor to booting from a ROM. The advanced toolset debugger cannot be used to debug such a program.

To make use of this facility, a different library is needed. The linker command line becomes:

```
ilink hello.tco -t425 -f cnonconf.lnk                  (UNIX)
ilink hello.tco /t425 /f cnonconf.lnk (MS-DOS and VAX/VMS)
```

The configurer can be bypassed and a special option on the collector used to direct it to build a bootable file from a linked unit.

The command line:

```
icollect hello.lku -t425 -t                            (UNIX)
icollect hello.lku /t425 /t             (MS-DOS and VAX/VMS)
```

will create the bootable file `hello.btl`.

This facility is provided for compatibility with previous versions of the toolset, but it will be discontinued in future releases.

4.3.7 Separate compilation

In practice, programs consist of several modules that must be combined to form a single program. An example in the directory `examples/simple` is contained in the three files `mainsep.c`, `hellosep.c` and `worldsep.c`. These all need to be compiled, one by one, using the command lines:

(UNIX)

```
icc mainsep -t425
icc hellosep -t425
icc worldsep -t425
```

(MS–DOS and VAX/VMS)

```
icc mainsep /t425
icc hellosep /t425
icc worldsep /t425
```

The object modules can now be linked together using a single command line:

```
ilink mainsep.tco hellosep.tco worldsep.tco -t425 -f cstartup.lnk
```
(UNIX)

```
ilink mainsep.tco hellosep.tco worldsep.tco /t425 /f cstartup.lnk
```
(MS–DOS and VAX/VMS)

to create the file `mainsep.lku`.

An alternative method of linking is demonstrated using a linker indirect file called `mainsep.lnk` and the command line:

```
ilink -t425 -f mainsep.lnk                              (UNIX)

ilink /t425 /f mainsep.lnk                 (MS–DOS and VAX/VMS)
```

to create the same file as before, viz. `mainsep.lku`.

The configuration description file `mainsep.cfs` which references the appropriate linked unit i.e. `mainsep.lku`, is now input to the configurer to produce a configuration binary file for the collector:

```
icconf mainsep.cfs
icollect mainsep.cfb
```

and the program bootable is called `mainsep.btl` and is ready for execution:

```
iserver -sb hello.btl                                   (UNIX)

iserver /sb hello.btl                      (MS-DOS and VAX/VMS)
```

to produce the greeting:

```
Hello World
```

5 Parallel processing

5.1 Introduction

Parallel processing is widely accepted as an important way of improving software performance on any given processor architecture. The transputer supports parallel processing directly by incorporating into its design a process scheduler which is responsible for scheduling parallel tasks, and by providing the means for connecting processors (transputer links) to create a multi–processor network.

Parallel programming is supported in the INMOS C toolset by extra library functions. These functions allow processes to be defined and created, to communicate with one another via channels and to synchronize using semaphores.

5.2 Abstract model

Parallel processing in transputer based systems is based on the idea of *Communicating Sequential Processes* (CSP) developed by Professor C.A.R. Hoare. (See '*Communicating Sequential Processes*' by C.A.R. Hoare, published by Prentice Hall International).

CSP is an abstract generalized model of concurrency based on the idea of independently executing processes exchanging data by synchronized communications. The model can be used to describe software applications in an intuitive way reflecting the parallelism of the real world.

Concurrent processing in INMOS C conforms to the CSP model. Concurrent C processes are independent, can be nested within each other, and are linked together by channels. Any C function can be defined as a concurrent process using a special set of functions provided in the runtime library.

Figure 5.1 illustrates the main elements of the CSP model. Processes can be nested within one another, and can communicate either unidirectionally (one process passing data to another) or bi–directionally (two processes exchanging data and working in a cooperative manner). In real applications processes normally communicate with at least one other process in the system.

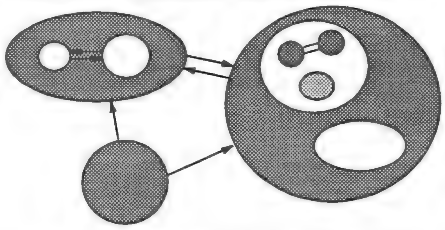


Figure 5.1 Communicating sequential processes

5.2.1 Processes

Processes are the main elements of the CSP model. A process describes the behavior of a discrete, separable component of an application; it may consist of other processes, sequential operations, or any combination of these. Applications can be broken down into any number of processes, and processes can be mapped onto a network of transputers.

5.2.2 Channels

Channels facilitate the communication between processes through which information and data are exchanged. Channels are point-to-point unidirectional connections, that is; they connect only two processes, and the transfer of data is one way. Processes which exchange messages and data with each other must do so via a pair of channels. Channels in real systems are often paired in this way to enable processes to cooperate in a task.

Channels have two functions. They provide the communication path between independently executing processes, and serve to synchronize the communication between the two processes. Processes which send data cannot do so until the receiving process is ready. Neither process can continue until the communication is completed. In this way synchronization between the two processes is assured; no data is passed until both partners in the operation are ready.

5.2.3 Semaphores

Support for semaphores, though not a part of the CSP model, is provided in the toolset for those who wish to develop parallel programs in the traditional manner. Semaphores are efficiently implemented within the toolset using channel functions, and are therefore subject to a slightly greater overhead than if the intrinsic synchronizing ability of channels were used directly.

5.3 Parallel processing and transputers

The transputer has been designed to support parallel processing and the construction of multiprocessor environments. The device architecture and instruction set reflect the CSP model and make it easy to implement in high level languages. INMOS C takes full advantage of this ability, providing a parallel programming environment optimized for the transputer, but retaining all the features of the standard language.

Each transputer separately supports parallel processing. A scheduling system built into the hardware of the processor automatically time-shares the CPU between processes and requires no extra input from the programmer. Processes can exchange data and synchronize their activity. Communication between processes is achieved via channels implemented as words in memory.

5.3.1 Multitransputer networks

Processes can also run on separate transputers and communicate with each other using channels implemented through transputer links. Each transputer contains (at most) four INMOS communication links through which processors exchange data and information. This ability to be cross-connected enables the transputer to be used as the basic component in the construction of processor networks. Specific arrangements of transputers can be designed for particular software tasks, and large networks of transputers can be used to build distributed processing supercomputers.

Compiled program and library modules are linked together to form '*linked units*'. Each linked unit is a C `main` program and is allocated to a particular transputer to run, by the configuration. Each transputer may run one or more linked units in parallel. The parallelism of the program as a whole is expressed by both the parallel processes within each compiled and linked unit and the fact that different parts of the program may be running in parallel on different transputers.

Chapter 6 explains how to write programs for a network of transputers and how to map the linked program units onto processors. While this chapter concentrates on the facilities available for utilizing several processes within a single C `main` program on a single processor.

5.3.2 Instruction set

Transputers have been designed to support the ideas of parallel processing and make them easy to implement in high level languages. There is direct support in the transputer instruction set for process control and management.

Process control

The transputer provides direct instructions for setting up, starting, pausing, and terminating parallel processes. Processes run at one of two priorities – high or low; high priority processes have priority access to the processor and will always be executed in preference to any low priority process running concurrently on the same processor.

Process selection

The transputer instruction set includes direct support for selection of the first ready input from a series of inputs, making polling of data channels unnecessary.

Process timing

The transputer contains high and low priority clocks, which can be used to implement delayed execution of processes. Specific instructions are provided to delay execution of a process for a specified time period, or until a specified time.

5.4 INMOS Concurrent C library

INMOS C takes full advantage of the advanced concurrency features of the transputer and, like the high level transputer language occam, implements the CSP model. Concurrency within a program is supported by library extensions consisting of three new data types and a set of library functions and macros. Together these implement the parallel model. A set of routines for synchronizing processes using semaphores is also provided.

The channel functions, provided by the library, support communication between processes running on either a single transputer or on a network of transputers.

5.4.1 Library support

The concurrency functions are accessed in the same way as other C library functions by including the appropriate header file in the program. Process, channel, and semaphore support functions are declared in three separate header files.

The concurrency functions are designed as a base set of functions which can either be used in their basic form or as building blocks for higher level routines. For example, a high level package might wish to implement features such as process multiplexing and complex channel protocols using functions from the basic set.

5.4.2 New data types

Three new data types complete the concurrency support. Data structures are used to hold data about processes and semaphores, and a pointer type is used to implement channels.

- `Process`. A structure type to hold information about a declared process.
- `Channel`. A data type used to implement channels. In accordance with the CSP model, channel variables represent unidirectional communication links between two processes. `Channel` is implemented by a pointer to type `volatile const void`.
- `Semaphore`. A structure type that holds information about a semaphore.

Parallel processes are created by linking a function definition to a pre–declared process structure, and are then initialized, started, and run using routines from the concurrency library. The header file `process.h` declares the process data type and library functions.

Channels between processes are created simply by declaring a variable of type `Channel` and initializing it by calling a library function. Channel input and output functions are then used to pass data. It is the responsibility of the programmer to ensure that data sent by one process is received by another; separate functions exist for input and output and the two must be paired for communication between two processes to take place. The header file `channel.h` declares the channel data type and library functions.

Semaphores are declared using either the semaphore initialization function or a macro that performs a similar action. Semaphores are then acquired and released by calls to two separate functions. Semaphores can be used to synchronize the activity of low with high priority processes. The header file `semaphor.h` declares the semaphore data type and library functions.

5.4.3 Concurrency functions

The concurrency functions implement the following parallel processing operations:

- Process setup, startup, and scheduling
- Select from several ready input channels
- Channel communication
- Semaphores.

The main parallel processing functions are declared in the two header files `process.h` and `channel.h`. Declarations of functions for semaphore handling can be found in `semaphor.h`. The following sections describe the process, channel, and semaphore functions.

5.5 Processes

When a program starts, there is a single main process in execution. Other processes, or threads, can be started on the same processor as the program executes. These other processes can be considered to execute independently of the main process, but share the processing capacity of the processor by time slicing.

A C function can be used as the entry point to a process provided that its first parameter is a process pointer. This will contain the pointer to its own Process structure. A process can share external data with other processes according to the usual C scoping rules.

The C function to be used as a parallel process must be defined with a given form of interface. This interface is made up of one fixed parameter followed by a number of non–fixed user defined parameters. The fixed parameter is the first parameter and is a pointer to a process structure `(Process *)`. The non–fixed parameters must be of types which are not subject to the default argument promotions (see section 4.2.3 of the *ANSI C Language and Libraries Reference Manual*). In addition the C function to be used as a parallel process may not take a variable argument list (i.e. an argument list terminated by `...)`.

For example:

```
void func(Process *p, int i, double d)
{
  /* function body */
}
```

Processes are created by one of the library functions `ProcAlloc` or `ProcInit`. They are started by one of the library functions `ProcRun`, `ProcPar` or one of their variants.

Here is an example of a simple program that starts a process, waits for the process to complete and then terminates. The sources for examples used in this chapter can be found in the `examples/simple` subdirectory.

```
/* This program starts a process to put out a simple */
/* message to stdout */

#include <stdio.h>
#include <stdlib.h>
#include <process.h>

/* The process is declared as a function */
void hello_proc (Process * p)
{
  p = p;      /* suppress compiler warning message */
  printf ("Hello, world.\n");
  return;
}

int main (void)
{
  Process * hello;

  /* Set up the new process */
  hello = ProcAlloc (hello_proc, 0, 0);
  if (hello == NULL)
  {
    printf ("Could not allocate process.\n");
    exit (EXIT_FAILURE);
  }

  /* Start it running */
  ProcRun (hello);

  /* Wait for the processes to join up again */
  ProcJoin (hello, NULL);

  /* Clean up all the space it used */
  ProcAllocClean (hello);

  exit (EXIT_SUCCESS);
}
```

5.5.1 Unused process pointer

The compiler generates a warning message indicating an unused process pointer each time a process pointer is declared as a parameter to a function, unless the parameter is used in some way. To prevent the message being generated the process pointer can be assigned to itself within the function using a statement of the form `p = p;`. Process code which does not assign the pointer in this way will still compile and run normally but the 'unused variable' message will be generated as the process is compiled.

Warning: The process pointer passed through to a function is used internally by the concurrency software and must never be changed. If it is modified in any way the results are undefined.

5.5.2 Process initialization

Two functions allocate and initialize parallel processes. A third function is provided to allow parameters to be altered in an existing process. The three functions and their parameters are listed below:

Function	Parameters
`Process ProcAlloc`	`(void (*func)(), int size, int nparam, ...)`
`int ProcInit`	`(Process *p, void (*func)(), int *ws, int wssize, int nparam, ...)`
`void ProcParam`	`(Process *p, ...)`

`ProcAlloc` reserves memory space for a process on the heap and initializes the `Process` structure using the lower level routine `ProcInit`. `ProcInit` can also be used directly to initialize a process for which the memory space has already been reserved by the programmer.

`ProcAlloc` takes a pointer to the function code, allocates a stack frame for the process, and sets up the function's parameters. The function is the code which will be executed by the process when it is started. `nparam` is the number of parameters to be passed to the function, excluding the compulsory process pointer.

The stack size is specified in the `size` parameter. If `size` is specified as zero a default stack size of 4K for 32-bit machines and 1K for 16-bit machines is used instead. If insufficient stack space is allocated for the required number of parameters, the stack is extended. `ProcAlloc` returns a pointer to the process structure (`Process*`).

Processes set up using `ProcAlloc` share the same global data space and therefore access the same static and external variables. Private data space for a process must be allocated using `auto` variables. In addition `ProcAlloc` uses the standard functions `malloc` and `free` to allocate and de-allocate space from the heap and as a result all C processes share the same heap space.

All calls to `ProcAlloc` should be followed by a check for successful allocation, and the NULL result (allocation unsuccessful) should be handled in an appropriate way.

The ancillary function **`ProcParam`** allows parameters to be changed in an existing (previously initialized) process. It must not be called after the process has started up.

`ProcInit` takes a pointer to an existing **`Process`** structure and a pointer to the stack space to be used. It then initializes the process structure and workspace for the function according to its workspace requirement and process parameters. Note: the pointer to stack space must not point to an area of memory within the stack space of an existing process, that is parallel process stacks must not nest. Stack space will usually be allocated using the functions **`malloc`**, **`calloc`**, **`realloc`** or will be declared as a static array.

`ProcInit` is the lower level routine used by **`ProcAlloc`**. `ProcInit` returns a value indicating success or failure. The number of words of parameters indicated by **`nparam`** excludes the compulsory process pointer.

Note: Processes must always be allocated before use. If this is not done the same memory space may be referenced on behalf of the same process. In this context allocation can be performed by **`ProcAlloc`** or `ProcInit`.

`ProcParam` can be used to modify the parameters of an already allocated process. It returns no result.

Note: Care should be taken when setting up processes and changing parameters in concurrently executing processes. If **`ProcAlloc`**, `ProcInit`, or **`ProcParam`** are used from two parallel processes to initialize the same process the results may be unpredictable because there may be contention for the process structure.

5.5.3 Freeing stack and workspace

Two functions **`ProcAllocClean`** and **`ProcInitClean`** are provided to free stack and workspace after a process has completed. The functions and their parameters are listed below.

Function	Parameters
void ProcAllocClean	(Process *p)
void ProcInitClean	(Process *p)

Function **`ProcAllocClean`** is used for processes initialized using **`ProcAlloc`**, and **`ProcInitClean`** for processes initialized using `ProcInit`.

5.5.4 Process termination

A process terminates by returning from the function started as a process. The function `void ProcStop (void)` will cause the process to wait indefinitely and never terminate.

5.5.5 Process execution (`process.h`)

A process executes a sequence of statements starting at the function which was associated with it via a **`ProcAlloc`** or `ProcInit` call and continues until it returns from that function or calls the function `ProcStop`. A process cannot terminate the whole program; only the main program can do that by means of the `exit` function or one of the INMOS extensions to `exit`.

Processes can be started asynchronously, or synchronously. Asynchronous initiation means that there is no implicit waiting for the process to terminate and the process continues independently of the execution of the process which initiated it. It might even outlast its initiator, although it is not recommended that it should continue executing beyond the termination of the main program. When a main function returns, or an `exit` function is executed, then the files are automatically closed and other tidying up operations performed. Processes that have not completed by this time can continue so long as they do not need access to the file system. The user is responsible for ensuring that asynchronous processes terminate.

Synchronous processes are started by a library function call and must complete before the library function will return. Hence, these processes cannot continue beyond the initiator. Several processes can be started at one time, in which case, they must all terminate before the initiator can continue. If the initiator continues then all the processes it initiated are guaranteed to have terminated.

Once a process has been started, either asynchronously or synchronously, it cannot be started again until it has completed its current execution. Any attempt to do this will result in the following fatal runtime error:

Fatal–C_Library–Attempt to start a process which is already running

Processes cannot be forcibly terminated during their execution; they can only terminate themselves.

Asynchronous processes

Asynchronous processes are started by one of the functions:

```
void ProcRun (Process * p);
void ProcRunHigh (Process * p);
void ProcRunLow (Process * p);
```

Any process can start any other process. The given process must have been initialized by either the `ProcAlloc` or `ProcInit` functions. `ProcRun` starts the process at the same priority as the currently executing process. `ProcRunHigh` starts one at high priority and `ProcRunLow` starts one at low priority.

Another process can wait for the termination of an asynchronous process by using one of the the functions:

```
int ProcJoin (Process * p1, ...);
int ProcJoinList (Process * * plist);
```

Any process can wait for any other asynchronous process (except the main process) to complete. The function `ProcJoin` will not return until all of the given processes have terminated. After it has returned, any memory dynamically loaded or allocated from the heap and used for the processes' stacks can be freed. Also any other resources that may have been held can be re-used. The list of process pointers in the parameter list is terminated by a NULL pointer.

`ProcJoinList` does an equivalent job when presented with a NULL terminated array of pointers to processes. If the array is empty, the function returns immediately.

Both these functions return zero if successful, and non-zero if the list of processes was empty.

A simple example of an asynchronous process was given in the previous section.

Synchronous processes

Synchronous processes are started by one of the functions:

```
void ProcPar (Process * p1, ... );
void ProcParList (Process * * plist);
void ProcPriPar (Process * phigh, Process * plow);
```

These each start a number of processes and wait for their termination before returning to the caller. `ProcPar` takes a list of process pointers ending in NULL and starts them all off. There is no guarantee as to the order in which they are started. All processes are started at the same priority as the caller.

`ProcParList` takes an array of process pointers, whose final entry is NULL. It behaves otherwise just like `ProcPar`.

`ProcPriPar` starts exactly two processes, one at high and one at low priority. It returns only after both have terminated. It is only possible to call `ProcPriPar` from a low priority process; a runtime error is reported if an attempt is made to call it at high priority.

Here is a simple example of using two processes to put out messages to `stdout` (standard out). The way the example is written does not specify the order in which the messages are put out. A method of ensuring the order is described in the next sections.

```
/* This program starts two processes, each of which */
/* writes a message. The processes are started twice, */
/* the second time swapping places in the parameter */
/* list of ProcPar. However, the order of the results */
/* to stdout is undefined.*/

#include <stdio.h>
#include <stdlib.h>
#include <process.h>

/* The first word */
void hello_proc (Process * p)
{
  p = p;
  printf ("Hello ");
  return;
}

/* The second word */
void world_proc (Process * p)
{
  p = p;
  printf ("world\n");
  return;
}

int main (void)
{
  Process * hello, * world;

  /* Set up the new processes */
  hello = ProcAlloc (hello_proc, 0, 0);
  world = ProcAlloc (world_proc, 0, 0);
  if ((hello == NULL) || (world == NULL))
  {
    printf ("Could not allocate process(es).\n");
    exit (EXIT_FAILURE);
  }

  /* Execute them both */
  ProcPar (hello, world, NULL);

  /* Try executing them in a different order in the list
*/
  ProcPar (world, hello, NULL);

  /* Clean up all the space it used */
  ProcAllocClean (hello);
  ProcAllocClean (world);

  exit (EXIT_SUCCESS);
}
```

Synchronizing between processes

Processes can synchronize their actions with each other either by means of semaphores or channels. Semaphores are the traditional way of coordinating activity in shared memory environments. Channels are used as a way of communicating values between two processes, but can also be used to synchronize processes because neither process can continue until the transfer has taken place.

Note: on transputers, there are special instructions that implement channels very efficiently, and semaphores are implemented using channels. If coordination is required between just two processes, then a channel is the fastest way to achieve this.

5.6 Channel communications (channel.h)

Channels are a way of transferring data from one process to another, or to provide a way of coordinating the actions of processes. If one process needs to wait for another to reach a particular state, then a channel is a suitable way of ensuring that.

If one process has exclusive access to a particular resource and acts as a server for the other processes, then channels can also act as a queuing mechanism for the server to wait for the next of several possible inputs and handle them in turn.

Any data can be passed down a channel, but the user must ensure that the processes agree a protocol in order to interpret the data correctly.

Channels only operate in one direction. So if two processes need to talk to one another, then two channels are needed, one in each direction.

Channels between processors are implemented by transputer links and will be described in the chapter on configuration. Channels used by processes on the same transputer are known as '*soft channels*'. Links between transputers are also known as '*hard channels*'.

5.6.1 Channel initialization

Channels can be declared like variables of any other data type, but in order to use a channel, it must be correctly initialized. This is done automatically for channels created by the configurer. For channels allocated by a C program this can be done with either of the following functions:

```
Channel * ChanAlloc (void);
void ChanInit (Channel *)
```

The function `ChanAlloc` allocates space for a channel from the heap and initializes the channel before returning. If the channel has already been allocated memory, it can be initialized by the function `ChanInit`.

The initial value for a channel is the constant named `NotProcess_p`, which is defined as a macro in `channel.h`. This value indicates that no process is waiting on the channel, either to read from it or to write to it. A channel must be initialized before any process attempts to read from or write to it. If it is to be passed as a parameter to two processes, then it should be initialized before it is passed, rather than letting one of the processes initialize it.

5.6.2 Channel output

To write a value to a channel, the following functions are provided:

```
void ChanOut (Channel *c, void *cp, int count);
void ChanOutChar (Channel *c, unsigned char ch);
void ChanOutInt (Channel *c, int n);
void DirectChanOut (Channel *c, void *cp, int count);
void DirectChanOutChar (Channel *c, unsigned char ch);
void DirectChanOutInt (Channel *c, int n);
```

The two functions `ChanOutChar` and `ChanOutInt` transfer a `char` or `int` respectively. `ChanOut` is a general transfer function that sends `count` bytes of data from array `cp` and can be used for any type of data.

When operating on channels that connect processes on the same processor, the functions with names beginning with '`Direct`...' are equivalent to the corresponding ones without: e.g. `DirectChanOut` is equivalent to `ChanOut`. There are differences, however, when used on channels that connect processes on different processors in a network. These differences are described in the chapter on configuration (chapter 6). The performance of channel functions is discussed in the document '*Performance improvement with the INMOS Dx314 ANSI C Toolset*'.

Each of these functions represents a single communication. The process will not continue until the transfer is complete.

5.6.3 Channel input

To read a value from a channel, the following functions are available:

```
void ChanIn (Channel *c, void *cp, int count);
unsigned char ChanInChar (Channel *c);
int ChanInInt (Channel *c);
void DirectChanIn (Channel *c, void *cp, int count);
unsigned char DirectChanInChar (Channel *c);
int DirectChanInInt (Channel *c);
```

The two functions `ChanInChar` and `ChanInInt` read a `char` or `int` respectively from a channel. `ChanIn` is a general transfer function that reads `count` bytes into the array `cp` and can be used for any type of data.

When operating on channels that connect processes on the same processor together, the functions with names beginning with '`Direct`...' are equivalent to the corresponding ones without. There are differences, however, when used on channels that connect processes on different processors in a network. These differences are described in chapter 6.

Each of these functions represents a single communication. The process will not continue until the transfer is complete.

Here is the example from section 5.5.5 with a synchronization channel inserted in order to ensure that the two message are printed in the same order every time it is run.

```
/* This program starts two processes, each of which */
/* writes a word of a message. The first one */
/* synchronises with the second after it printed */
/* its word.*/

#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>

/* The first word */
void hello_proc (Process * p, Channel * c)
{
  p = p;
  printf ("Hello ");
  ChanOutInt (c, 1);     /* tell the second part to go */
  return;
}

/* The second word */
void world_proc (Process * p, Channel * c)
{
  int k;

  p = p;
  k = ChanInInt (c);
  printf ("world\n");
  return;
}

int main (void)
{
  Process * hello, * world;
  Channel * sync;

  /* Set up the communication channel */
  sync = ChanAlloc ();

  /* Set up the new processes */
  hello = ProcAlloc (hello_proc, 0, 1, sync);
  world = ProcAlloc (world_proc, 0, 1, sync);
  if ((hello == NULL) || (world == NULL))
  {
      printf ("Could not allocate process(es).\n");
      exit (EXIT_FAILURE);
    }

    /* Execute them both */
    ProcPar (hello, world, NULL);

    /* Try executing them in a different order in  */
    /* the list*/
    ProcPar (world, hello, NULL);

    /* Clean up all the space it used */
    ProcAllocClean (hello);
    ProcAllocClean (world);

    exit (EXIT_SUCCESS);
  }
```

5.6.4 Reading from several channels

There are many cases where a process listens to several channels and wishes to detect which one has data ready first. This is done using one of the following functions that determine which alternative input to read:

```
int ProcAlt (Channel * c1, ...);
int ProcAltList (Channel * * clist);
int ProcSkipAlt (Channel * c1, ...);
int ProcSkipAltList (Channel * * clist);
```

These all take parameters that are lists of channel pointers, terminated by a NULL pointer. `ProcAlt` and `ProcAltList` both return an index (starting at zero) of a channel that is ready to transfer data.

A simple example of a server that converts its input to upper case and returns it down the corresponding response channel is shown below.

```
void upserve (Process * p, Channel * ins[], Channel * outs[])
{
  size_t i;
  char data;

  p = p;
  /* Loop forever, this process never terminates */
  for (;;)
  {
    /* wait for an input */
    i = ProcAltList (ins);

    /* read and convert data */
    data = ChanInChar (ins[i]);
    data = toupper(data);

    /* output it along corresponding channel */
    ChanOutChar (outs[i], data);
  }
}
```

A complete program that uses this is found in the directory **examples/simple** under the name of '**upserve.c**'.

ProcAlt and **ProcAltList** will wait until a channel is ready.

The variants of the functions **ProcSkipAlt** and **ProcSkipAltList** can be used not only for selecting between alternate channels, but to detect whether any of them are ready. They return with the value '**-1**' when none of the channels are ready; they otherwise act like **ProcAlt** and **ProcAltList** respectively.

5.7 Semaphores (semaphor.h)

In C, all processes have access to all external and static variables that are in scope. In order to prevent contention when updating these shared variables, semaphores can be used. They are used, for example, by the C runtime library to protect access to the heap and file system.

Semaphores can be defined to allow a given number of processes simultaneous access to a shared resource. The number is determined when the semaphore is initialized. When that number of processes have acquired the resource, the next process to request access to it, will wait until one of those holding it relinquishes it. Thus semaphores operate as a queuing mechanism, where the order of acquisition of the resource, is strictly the order of requesting it.

Semaphores can protect a resource only if all processes that wish to use the resource also use the same semaphore. It cannot protect a resource from a process that does not use the semaphore and accesses the resource directly.

Typically, semaphores are set up to allow at most one process access to the resource at any given time.

The semaphores can be used by either high or low priority processes for the same resource. The priority does not affect the position the process will appear in the waiting queue if it must wait. The order of access to the semaphore is strictly chronological.

Semaphores must first be created and initialized. The following are provided for this purpose:

```
Semaphore * SemAlloc (int initvalue);
void SemInit (Semaphore * sem, int initvalue);
SEMAPHOREINIT (int initvalue);
```

SemAlloc allocates space for a semaphore from the heap and initializes it with the number of simultaneous accesses it will allow. **SemInit** is used when the semaphore is obtained in some other way, e.g. declaring it as a variable, but it must be initialized before it can be used. The macro **SEMAPHOREINIT** can be used to initialize a static variable of type **Semaphore**.

Access to a semaphore is controlled by the functions:

```
void SemWait (Semaphore *);
void SemSignal (Semaphore *);
```

When a process needs to acquire a semaphore, then **SemWait** is used. If the number of processes that can simultaneously use it is exhausted, then the requesting process will wait until the number drops below the maximum.

When a process has finished using a resource, then it must use **SemSignal** to release it for another process. The first waiting process (if any) will then be given access to the semaphore.

5.8 Timers and delays

The transputer has two on–chip timers, one for low priority and one for high priority processes. The low priority one runs at a rate of 15625 ticks per second, and the high priority one at one million ticks per second. The machine instruction to read a timer reads the timer whose priority is the same as the priority of the executing process. Thus, high priority processes read the high priority timer; low priority processes read the low priority timer.

The value of a timer (or clock) can be held in an **int**, and there are a number of functions that can manipulate these values:

```
int ProcTime (void);
int ProcTimePlus (const int t1, const int t2);
int ProcTimeMinus (const int t1, const int t2);
int ProcTimeAfter (const int t1, const int t2);
```

ProcTime reads the timer for the current priority. **ProcTimePlus** adds two timer values together; **ProcTimeMinus** subtracts the second value from the first.

ProctimeAfter determines whether the first time is after the second time. One time is considered to be after another if the one is not more than half a full timer cycle later than the other. Half a full cycle is 2^{31} ticks on a 32-bit transputer or 2^{15} ticks on a 16 bit transputer. The function returns the int value '**1**' if **t1** is after **t2**, otherwise it returns zero.

5.8.1 Control of processes by timers

A process can be made to wait for a certain length of time as measured in ticks of the timer. The functions:

```
void ProcAfter (int t);
void ProcWait (int t);
```

Both functions wait for a period of time and then return; **ProcAfter** waits until the given absolute reading of the timer is reached. If the requested time is not after the present time, then the process does not wait.

ProcWait suspends the current process until the given time has elapsed, i.e. it delays execution for the specified number of timer ticks. If the time given is negative, no delay takes place.

There are also two functions that allow a process to select from a list of input channels with a time-out. If no channel becomes ready within a given time, then the function returns and the process can continue execution. These functions are:

```
int ProcTimeAlt (int time, Channel *c1, ... );
int ProcTimeAltList (int time, Channel * * clist);
```

These functions return on the occurrence of the earlier of either an input becoming ready on any of the channels or the expiry time. The time delay behaves like `ProcWait`. The value '`-1`' is returned if the time expires with no channel becoming ready.

5.9 Other process facilities

If a process needs to know whether it is running at high or low priority, it can use the function:

```
int ProcGetPriority (void);
```

which returns zero for high priority and one for low priority.

Sometimes, a process needs to forcibly give up control of the CPU so that another process can execute, i.e. terminate the current time-slice. The function:

```
void ProcReschedule (void);
```

does just that. It provides a clean way of suspending execution of a process in favor of the next processor on the scheduling list, but without losing priority. The process which executes `ProcReschedule` is added to the back of the scheduling list.

6 Configuring transputer programs

This chapter examines how to make use of more than one transputer in a single application. If several transputers are used, it is possible to gain performance from the fact that the individual processors execute in parallel.

This chapter describes how to configure programs to run on transputer networks; the chapter is divided into four main sections as follows:

- Configuration basics
- Configuration language
- Further considerations
- Examples

In chapter 9, more advanced configuration techniques are described.

6.1 Configuration basics

This section introduces the concept of configuration and takes the user through each stage of configuration, ending with an example of the complete process. The section covers the following topics:

- Introduction to configuration
- Hardware description
- Software description
- Mapping description
- Types of main program
- Access to interface parameters
- Example configuration

6.1.1 Introduction to configuration

Each transputer has up to four hardware links. These links can be connected to one another to form communications links between transputers. When there are several such transputers connected in this way, they are known as a *network*, and the individual transputers are called *nodes* of the network.

In order to make use of the network, the application must be divided into several parts and these parts placed on the nodes of the network. C applications are broken down into a number of smaller programs, each having a C `main` function. Each program is then placed on a specified transputer. Programs communicate via channels. The software channels, between processors, are implemented on the hardware links.

To describe how the application is to be spread out among the nodes of the network, a language called the *configuration language* is used. The user must describe:

- What the nodes of the network are and how they are connected.
- What the software looks like and how its various parts are connected via channels.
- Which bits of the software are to be placed on which bits of hardware.

In this way, it is possible to control which parts of an application are to placed on which nodes of the network, thus ensuring that, for example, computationally expensive parts are on T805s and that device controllers are on T225s.

It is assumed here that a network is initialized by booting the application down one of the links; the link connected to the host computer. This link is known as the '*bootlink*' and is typically also used for access to the host services by the application. The connections between the network and the outside world e.g. the host or peripheral devices are known as *edges*.

A diagrammatic representation of the configuration process is given in figure 6.1.

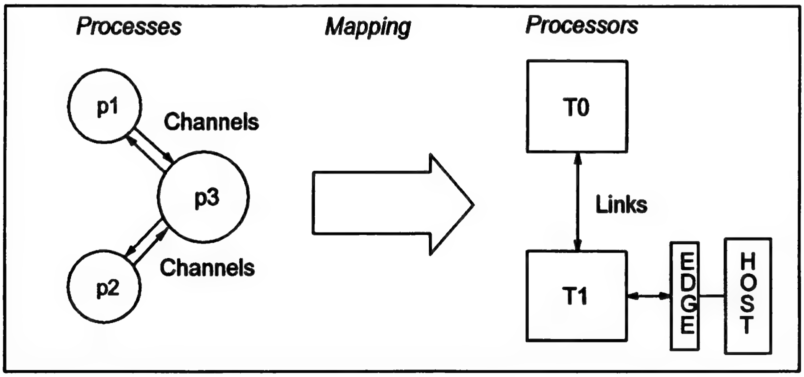


Figure 6.1 The configuration model

6.1.2 Hardware network description

The first part of the configuration description is that for the hardware network. It defines what nodes are present and how they are connected by means of their links. A simple example was shown in *'Getting Started'* – chapter 4, defining a single node connected to the host.

Hardware networks consist of nodes of type `processor` connected by links. The hardware description contains declarations of processors in the network together with the connect statements that join them by their links.

Processors have the following user-definable attributes:

`type`	The processor type. INMOS standard transputer types are predefined.	Mandatory. Must be specified before any links can be used.
`memory`	The amount of memory available to the processor.	Mandatory.
`reserved`	Specifies the size of memory in bytes, which cannot be used by the configurer to place user and system processes.	Optional.
`router`	This attribute has the following sub–attributes: `linkquota`, `routecost` and `tolerance`. It is used to determine virtual routing criteria.	Optional.

Attributes can be specified in the processor declaration. A typical processor declaration could be:

```
processor(type = "T414", memory = 1M) root;
```

Processor attributes do not have to be specified in one declaration. Once a processor has been declared, for example as above, further attributes can be added later on in the hardware description, e.g.

```
root (reserved = 6K);
```

The configurer will fail, however, if any of the mandatory attributes are omitted from the configuration description.

The order in which process attributes is declared is also not significant, except that the processor `type` attribute must be defined before the processor links can be used.

Processors are connected to each other by processor links. The number of links is defined by the processor type.

Processor links

Links are special attributes of processors that are predefined by the configurer. Link attributes cannot be modified by the user.

The number of links for each type of processor is predefined within the configurer via the processor `type` attribute. The value of this attribute is defined for all INMOS transputer types listed in the configuration defaults include file `setconf.inc` which is read by the configurer at startup. Definition of the `type` attribute therefore defines the number of links available to a particular processor. Link numbers begin at zero.

Links are referenced using the dot notation and can be treated as arrays. For example, they can be subscripted as though they are arrays:

```
connect root.link[2] to transputer1.link[0]
```

and the `size()` function can be used to determine the number of links on a processor:

```
size(root.link)
```

Links can be connected to the links of other processors and mapped with software channels. Links may only be connected to one other link (or network edge, see below). Links can also be left unconnected.

Defining new processor types

Processor types can be defined for later use in a configuration. In the following example the processor type `T800` is first defined using the predefined type `processor` and then used to define a further processor type called `B405` to be a `T800` equipped with a set amount of memory. (The `B405` definition corresponds to the INMOS *i*q systems IMS B405 TRAM product.)

```
define processor(type = "T800") T800;
define T800(memory = 8M) B405;
```

Certain processor types are predefined in the configuration by the automatic inclusion of the `setconf.inc` file at configurer invocation. The file provides definitions of all transputer types manufactured by INMOS along with some other predefinitions.

Predefined types can be used as though they are part of the language and do not need to be referenced by an `include` statement. The definitions are listed in section B.3.2.

Edges

Edges are hardware network variables which bring transputer links out of the network for connection to the outside world, that is, to external devices or other networks. They are directly analogous to channel edges in software networks. Edges have the same characteristics as processor links. Edges can only be connected to other transputer links.

In the following example an edge is declared which allows a processor in a hardware network to input data from an A-to-D convertor:

```
T414 data_handler;

edge a_to_d;

connect data_handler.link[1] to a_to_d;
```

Arrays of edges can be useful constructions. In the following example an array of edges is declared for a series of sampling lines and then connected to three links of a processor which logs the data from each line. The remaining link is used to boot the processor.

```
edge samplers[3];

rep i = 0 to 2
  connect data_logger.link[i] to samplers[i];
```

Host edge

A special edge called `host` is pre–declared in the configurer defaults file and can be used without declaring it in the configuration. In all configurations that will be loaded from a host system, there must be one, and only one, processor link connected to `host`.

The `reserved` attribute

Specifies the size of memory, in bytes to reserve from **MOSTNEG INT**, which cannot be used by the configurer to place user and system processes. If no reserved attribute is defined then the region of memory available to the configurer is defined by the value of the `memory` attribute minus the **LoadStart** offset for the processor.

The syntax for the `reserved` attribute is as follows:

processor`(reserved = `*exp*` );`

where: *processor* is the name of a declared processor or processor type.

exp is an expression.

The syntax of the configuration language is given in appendix B.

The use of the `reserved` attribute is described in section 9.1.

The `router` attribute

The `router` attribute and its sub–attributes `linkquota`, `routecost` and `tolerance` enable the user to partition a network and weight the use of particular virtual routing paths through the network. The attributes are optional and do not have to be used for virtual routing to be enabled. Their use is described in chapter 9.

6.1.3 Software network description

Typically, the software is described in manner very similar to the hardware. The processes within an application can be thought of like the nodes of a network, and the channels that connect them are akin to the hardware links. However, there are many more possible variations for a given process than there are for transputers.

When preparing an application for a network, each software process is constructed as though it were a separate program. All the source files for it are compiled, the object modules are then linked to form a file known as a *linked unit*. The linked unit embodies the program in a way that can be used as a process in a network.

The software network is composed of nodes of the predefined type `process` connected by input and output channels. The software description consists of a series of process declarations that define the network's interface with the outside world, and the connections between them. A separate statement is used to assign linked modules to the software processes.

A typical process declaration would be as follows:

```
process(stacksize=2K, heapsize = 16K,
        interface(int count, input in)) p;
```

Process attributes

Each process possesses a set of attributes some of which must be given values in the process declaration; others are optional with built-in defaults. The attributes of a process are:

`stacksize`	The size of stack required for the process in bytes.	Mandatory for C programs
`heapsize`	The size of heap required for the process in bytes.	Mandatory for C programs
`interface`	The list of parameters to the process.	Mandatory if external communications are required.
`priority`	The execution priority for the process. Priority can be defined `HIGH` or `LOW`. Default is `LOW`.	Optional
`order`	The ordering of program segments in memory. Segments which can optionally be ordered are: `code`, `stack`, `static`, `heap`, `vector`. The default for each segment is 0.	Optional
`location`	The re–location of program segments in memory. Segments which can optionally be re–located are: `code`, `stack`, `static`, `heap`, `vector`.	Optional
`nodebug`	For use with the *Advanced Toolset* only. Indicates process is not to be debugged. Takes the value `TRUE` or `FALSE`; the default is `FALSE`.	Optional
`noprofile`	For use with the *Advanced Toolset* only. Indicates process is not to be profiled. Takes the value `TRUE` or `FALSE`; the default is `FALSE`.	Optional

Process attributes do not have to be specified in one declaration. Once a process has been declared e.g.

```
process(stacksize=2K, heapsize = 16K,
        interface(int count, input in)) p;
```

Further attributes can be added later on in the software description, e.g.

```
p (priority = HIGH);
```

The configurer will fail, however, if any of the mandatory attributes are omitted from the configuration description.

The order in which process attributes is declared is also not significant.

- **Program segment sizes**

  For C programs `stacksize` and `heapsize` are mandatory. The sizes of the `code`, `vector`, and `static` program segments are fixed by the compiler or linker. For C programs the vector program segment is unused.

- **Interface attribute**

  The `interface` attribute is mandatory if the process needs to communicate with another process or an edge. It defines the way in which the process interacts with the outside world and with other processes via a set of parameters. The parameters can be read by a source program using the library function `get_param`.

Parameters to the `interface` attribute can be input channels, output channels, simple data types, or arrays of these. Permitted data types for the parameters are `int`, `char`, `float`, and `double`. Strings are defined as arrays of characters and may be initialized by a quoted string constant.

Each input channel can only be connected to an output channel on another process. The rules for connecting channels to software network edges are described below under the heading '*Edge connections*'.

Values can be defined for interface parameters either by assigning a value in the process declaration or by a separate statement. For example:

```
process (interface (int count = 10,
                    input command,
                    output result)) task;

/* count defined at declaration as 10 */

task (count = 100);

/* count redefined as 100; this value for count
   remains in force until redefined again */
```

Values given to parameters may also be derived from a replicator count using an expression including the index variable.

- **Array parameters**

When assigning parameters which are numeric arrays it is not possible to assign individual elements of the array but only the complete array. For example:

```
x(y = {0,1,2,3})
```

- **`get_param` function**

A special library function `get_param` is provided to receive the process interface parameters within the C program. The function returns pointers to the parameters and is used to retrieve them from the configuration code.

Details of the function's operation can be found under the function description in the *ANSI C Language and Libraries Reference Manual.*

- **Host server channels**

  For C programs which use the host file server, the first parameter must be an input channel and the second parameter must be an output channel. These two channels are used by the *full* C library to contact the host server. They can be obtained by use of the `get_param` function call , but it is recommended that you do not do so. The function `server_transaction` is provided if you wish to contact the server directly.

- **Execution priority attribute**

  Initial runtime priority for the process can be set high or low by specifying `priority=HIGH` or `priority=LOW`. The default is `LOW` priority.

- **Segment ordering attribute**

  The order in which the program segments are automatically placed in transputer memory can be changed by specifying an ordering priority for each or any of the segments. The default is no segment ordering.

  The syntax for the `order` attribute is as follows:

  process `(order (` {*segment* = *value*} `));`

  where: *process* is the name of a declared process or process type.

  segment is one of: `code  stack  vector  static  heap`

  value is the ordering priority and can take any integer value, the default is '0'. Positive values indicate placement higher in memory and imply lower speed access; negative values indicate lower memory placement and imply higher speed access. The lower the placement, the greater the chance that code or data will be placed in on-chip RAM, which has the fastest access.

  If no order is specified a default segment ordering is applied, see section 9.1.1. Further details about ordering can be found in chapter 9.

  Example:

  ```
  p (order (code = -1000));
  ```

  In this example the `order` attribute will override the default segment ordering and cause the user's code segment to be placed lower in memory than the stack segment. (Provided no further segment ordering overrides this statement).

- **Segment re–location attribute**

  The `location` attribute enables the user to specify a start address for individual program segments. All program segments not specifically addressed using the `location` attribute will be automatically placed by the configurer.

The syntax for the `location` attribute is as follows:

process `(location ( {`*segment* `=` *address*`} ));`

where: *process* is the name of a declared process or process type.

segment is one of: `code stack vector static heap`

address is expressed as an integer value.

Code and data re–location is described in section 9.1.

- **Support for the Advanced Toolset**

  Two attributes are included for use with the *Advanced Toolset* product only. They are the `nodebug` and `noprofile` attributes. If set equal to `TRUE` for a process, the advanced debugging and profiling tools supplied with the *Advanced Toolset* will ignore that process. The attributes have no affect on the functioning of `idebug` or any other tool supplied as part of the current toolset.

Defining new process types

Specific process types can be defined and used later in the program with different actual parameters. For example, the following code defines a process type `filter` which is later used to declare three filter processes with different values for the `cutoff` parameter.

The processes are configured to form a pipeline starting and finishing at the host. The connection statements linking interface channel variables to host channel edges are shown for completeness. The host channel edges are assumed to have been defined earlier in the program.

```
define process (stacksize = 2K, heapsize = 16K,
         interface (input in, output out,
                    int cutoff)) filter;

filter x, y, z;

x(cutoff = 10);
y(cutoff = 20);
z(cutoff = 10);

connect x.in to from_host;
connect y.in to x.out;
connect z.in to y.out;
connect z.out to to_host;
```

Attributes can also be activated for specific instances of a type by specifying them within the declaration. In the following example the priority is defined for a single instance of the `worker` type; no other instances are affected and all other attributes are unchanged:

```
worker (priority = HIGH) ant;
```

Extra attributes can also be supplied in process definitions or in attribute assignment statements. For example:

```
define worker (heapsize = 20K) small_worker;
small_worker drone;

worker bee;
bee (heapsize = 200K);
```

Input and output channels

Processes which cooperate with each other and exchange data are connected by channels of types `input` or `output`. These channels are equivalent to channels in source code programs.

To send or receive data processes must declare input and output channels. The sending process must declare an output channel and the receiving process an input channel. The configurer will automatically route channel communication between processors provided input/output channels have been declared and connected correctly. It is also possible to specifically place channels on links by the `place` statement.

In the following example a host monitor process `host_process` sends data via the output channel `out` to the application process `p`, which receives it on the input channel `in`.

```
process(stacksize = 2K, heapsize = 16K,
        interface (int count, input in)) p;

process(stacksize = 2k, heapsize = 16K,
        interface (output out)) host_process;

connect p.in to host_process.out;
```

Further information about the use of channels is given in section 6.1.4.

Edge connections

The software network can be connected to the outside world by channel edges. Channel edges are input and output channels declared within the software description and connected to input and output channels of a process. A process can then import or export data from the software network. Edges are commonly used for interfacing i/o processes to the host server.

Unlike channels between processes, connections between edges and process channels must be of the same polarity, that is, an input edge must be connected to an input channel and an output edge to an output channel. This preserves the direction of the channel parameter. For example, the following code creates an interface between `p` and the host server:

```
process (stacksize = 2K, heapsize = 16K,
         interface (input in, output out)) p;

input from_server;                /* input edge */
output to_server;                 /* output edge */

connect p.in to from_server;      /* input path */
connect p.out to to_server;       /* output path */
```

6.1.4 Mapping description

Having defined both the hardware and software networks, it is now necessary to say which process is to be placed on which processor and to assign actual code modules to processes.

The mapping description defines how an application, described as a software network of processes connected by channels, is mapped onto a hardware network of processors. This assignment is performed by the `place` statement:

`place` *process* `on` *processor;*

`place` *channel* `on` *link;*

where: the placement of processes on processors and channel edges onto hardware edges is mandatory but placement of channels, which connect processes, on links is optional.

Placement of processes

The placement of processes is simply the assignment of application processes to the target hardware processors on which they are required to run. It is mandatory to make such an assignment for each process in the software network.

For example if the process `filter` is required to run on processor `root`, the following `place` statement would be used:

```
place filter on root ;
```

Placement of channels

Channels are unidirectional, point–to–point connections which may be implemented in one of four ways:

- *Soft* channel – a channel which communicates between processes running on the same processor.
- Channel *edge* – a channel which provides communication between the network and the outside world.
- *Direct* channel – one of up to two channels (one in each direction) placed on a single link between adjacent processors.
- *Virtual* channel – a channel placed on on a *virtual* link.

No further action is required at configuration time to define or place the *soft* channels within an application; they are fully defined by the software itself.

Channel edges *must* be placed on a *hardware edge*. This involves specifying the name of the channel connection which is the be placed on a named *hardware edge* e.g.

```
    place from_server on host;
```

All other channels on a network may be implemented as either *direct* channels or *virtual* channels. By default the configurer *automatically* places software channels on links using the placement of processes on processors and channel edges on hardware edges as a guide.

The configurer can implement many channels over a single hardware link as well as channels between non-adjacent processors; channels implemented in this way are known as *virtual channels*. They are implemented by software virtual routing processes added automatically, as required, by the configurer.

Direct channels occur when only one or two channels (one in each direction) are placed on a link between adjacent processors. Direct channels may be automatically allocated by the configurer or the user may specifically place up to two channels on a named link. For example, a channel *edge* is an example of a mandatory direct channel. **Note:** when software virtual routing is required by the configuration, the configurer may override the user's specification of a direct channel, provided that it is not an *edge* channel.

Virtual channels enable an application program to run on most network topologies irrespective of the number of physical links connecting processors. The configurer can form virtual channels that span up to 24 hops across the target network. (A 'hop' is when a processor is required for routing a channel, zero hops implies that no processors were required to route the channel). Should the configurer fail to implement a long distance connection in a very large network, it will generate an error message. Chapter 9 provides further information about routing channels.

Virtual channels are unidirectional and synchronized and are implemented by means of *virtual links*. A virtual link can be thought of as a bi-directional virtual connection between two processors, providing a communication path sufficient for two channels (one in each direction) and the appropriate synchronization signals.

Explicit placement of channels on links using *direct* channels is only required when connecting channels to hardware edges or where links are used for special purposes. For example, connection to a device, or where an application uses input and output channels separately, as in software implementations of high-speed links. In certain performance critical applications it may also be important to avoid the overhead incurred when using virtual channels.

A link may carry both explicitly placed channels as well as virtual channels and a pair of virtual channels (one in each direction) may be routed by the configurer via different physical links.

The configurer also performs automatic placement of one end of a channel connection if the other end is explicitly placed.

In general channels should not be explicitly placed on links, unless they are edge connections. This enables the configurer to implement channels where applicable using routing and multiplexing software.

Attention: If it is essential that the configuration does not use any virtual routing, e.g. for performance reasons, the `icconf` 'NV' command line option should be used. This disables the configurer from using the virtual routing processes. (The configurer will fail if configuration is not possible, in which case the configuration should be modified to ensure that all channels can be placed). The bootable file generated will be smaller when the virtual routing processes are not included.

Predefined connection names

Predefined `connection` names can also be used to place named channels on named links. For example:

```
connection root_link0_host;
connect root.link[0] to host by root_link0_host;

connection master_ts_to_server, mastet_fs_from_server;
connect master.ts to to_server by master_ts_to_server;
connect master.fs to from_server by master_fs_from_server;

place master_ts_to_server on root_link0_host;
place master_fs_from_server on root_link0_host;
```

Assigning code to processes

Code is assigned to specific processes by means of the `use` statement. This associates a specific object file with a process. The object file must be a linked unit, for example:

```
use "filter.lku" for filter
```

Note: for C applications, each linked unit must be a complete C program containing a C `main` function.

In the following example a linked unit `filter.lku` is assigned to each of three processes:

```
filter x, y, z;

use "filter.lku" for x;
use "filter.lku" for y;
use "filter.lku" for z;
```

Linked units can also be associated with process *types*. This allows the same code to be assigned to several processes in a single statement. For example:

```
filter x, y, z;

use "filter.lku" for filter;
```

Mapping example

In the following example the software network consists of an i/o process `host_process` and a worker process `task` connected by the channels `in` and `out`. Two processors `root` and `servant` joined by a single link connection form the hardware network. The root processor is assumed connected to the host via link zero (in the hardware description). The code for the application comprises the two linked units `master.lku` and `slave.lku`.

The following mapping description places each process on one of the processors and assigns host server channel edges to the host edge link:

```
place fs on host;
place ts on host;

place host_process on root;
place task on servant;
```

If it was required to explicitly place the channels `in` and `out`, and given the following channel and link connections:

```
connect host_process.out to task.in;
connect host_process.in to task.out;

connect root.link[1] to servant.link[0];
```

the mapping would be as follows:

```
place task.in on servant.link[0];
place task.out on servant.link[0];

place host_process.in on root.link[1];
place host_process.out on root.link[1];
```

The mapping is completed by assigning the linked units to the software processes:

```
use "master.lku" for host_process;
use "slave.lku" for task;
```

6.1.5 Types of main program

The C language has pre–defined functions that require access to a host system, e.g. to use a file system. A transputer, however, is just a node in a network and can communicate only along its links. Thus, in order to provide access to host system facilities the C program must have some means of accessing the host system. This is done by means of host channels which must be supplied to the C program via the `interface` attribute of the process.

In the simplest scenario, a program is booted down a link by the `iserver`, a program that executes on the host. The `iserver`, after loading the network, waits for a response from the application and performs the services it is asked for.

Typically, however, only one of the processes in a network can be connected to the host, and hence to the `iserver`. So only this process can have access to the host's facilities. This process must be linked with the *full* library, by using the file `cstartup.lnk` when linking, which incorporates all the host access functions and can implement the full range of functions in the C language.

Without access to a host system a process cannot use any functions that require host facilities, e.g. those functions using the `FILE` type in C. In order to pick up a suitable library (the *reduced* library) for this case, the program must be linked with the file `cstartrd.lnk`, which incorporates a reduced set of library functions. In this case, the runtime system does not attempt to access a server.

In the reduced case, all communications with the environment are through channels, via the parameters that are passed in through the `interface` attribute of the process.

6.1.6 Access to interface parameters

When writing a program for a network node, the parameters from the configurer are made available by means of the function:

```
void * get_param (int n);
```

This function returns the *n*'th parameter as a pointer to the value (starting from 1). If the parameter is a channel, then it can be converted directly to a value of type

`Channel *`. The details of this function are provided in the *ANSI C Language and Libraries Reference Manual*.

In the case of a *full* library, there must be two channels for communicating with the server. These are the first two parameters in the interface description. The first is the input channel from the server to the program and the second is the output channel from the program to the server. The results are undefined if these are not provided.

Note: that the `main` function in a C program has parameters that are defined by the C language. In the *full* case, these parameters are acquired from the host's command line. In the *reduced* case, the value of `argc` is zero. These parameters to `main` are not related to the parameters available from the configurer by means of the `get_param` function.

6.1.7 Example configuration

The following example prints the 'Hello World' greeting. The first word comes from a process running on the root processor attached to the host directly. The second word is derived from a process running on another processor attached only to the root. The source for the example can be found in the `examples/simple` subdirectory.

The controlling process is as follows:

```
#include <stdio.h>
#include <channel.h>
#include <misc.h>

/* This program is loaded onto the root processor of a        */
/* two-proc network It puts out the first word of a message,  */
/* and then acquired the second word from a channel, and      */
/* then prints it.                                            */

int main()
  {
    int i;
    Channel * in_chan;      /* channel to get second word from */
    char buffer[101];       /* buffer to hold second word */

    in_chan = (Channel *) get_param (3); /* get the channel */
    printf("\nHello ");                  /* output first word */

    for (i = 0; i < 100; ++i)
    {
      buffer[i] = ChanInChar (in_chan);
      if (buffer[i] == '\0')
        break;
    }
    printf ("%s\n", buffer);
  }
```

This program is converted to a linked unit by following standard toolset development steps as demonstrated in the '*Getting Started*' chapter. e.g.

(UNIX)

```
icc hello.c -t5 -o hello.tco
ilink -t5 hello.tco -f cstartup.lnk -o hello.lku
```

(MS–DOS and VAX/VMS)

```
icc hello.c /t5 /o hello.tco
ilink /t5 hello.tco /f cstartup.lnk /o hello.lku
```

The transputer class T5 enables the program to run on a T400, T425 or T426 transputer. (Further information about transputer targets can be found in appendix B of the *ANSI C Toolset Reference Manual*).

The secondary process just writes down a channel, thus:

```
#include <string.h>
#include <channel.h>
#include <misc.h>

/* This program is loaded onto a second node and merely     */
/* transfers a single word to another process for printing. */

int main()
  {
    int i;
    Channel * out_chan;                /* channel for output */
    static char * word = "World";

    out_chan = (Channel *) get_param (1); /* get the channel */
    for (i = 0; i <= strlen(word); ++i)
    {
      ChanOutChar (out_chan, word[i]);
    }
  }
```

The secondary process has no access to the host server; hence it is linked with the reduced library by using the startup file `cstartrd.lnk` e.g.

(UNIX)

```
icc world.c -t8 -o world.tco
ilink -t8 world.tco -f cstartrd.lnk -o world.lku
```

(MS–DOS and VAX/VMS)

```
icc world.c /t8 /o world.tco
ilink /t8 world.tco /f cstartrd.lnk /o world.lku
```

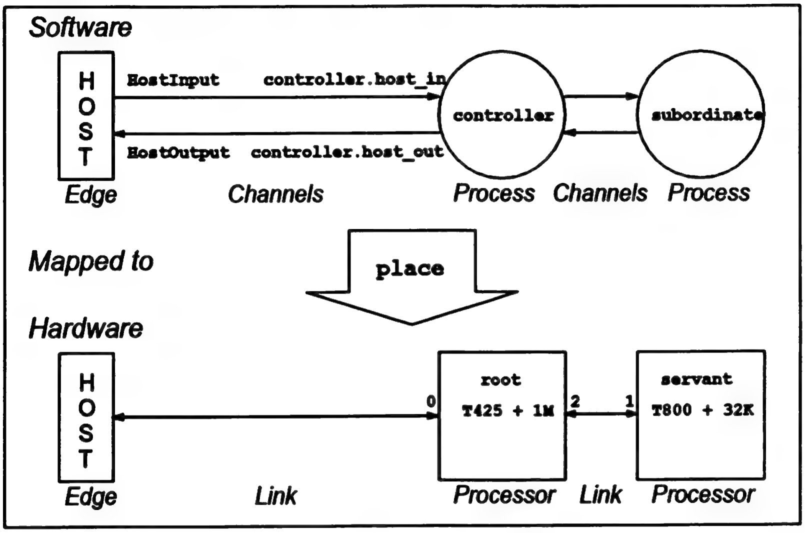
In this case transputer class T8 is used, enabling the program to run on a T800, T801 or T805 transputer.

The configuration file, which can be found in the `examples` directory under the name '`twin.cfs`' is as follows:

```
/* (c) Copyright INMOS Limited  1992. */

/* This defines a network of two processors:

   HOST ---- T425+1M ---- T800+32K

*/
/* Define the root processor */

T425 (memory = 1M) root;

/* and the servant processor */

T800 (memory = 32K) servant;

/* and their connections */

connect host to root.link[0];
connect root.link[2] to servant.link[1];

/* The software is similar */

/* The controller will reside on the root, and it has access */
/* to the host's services. Hence, it must have as its first  */
/* parameter the channel for input from the host, and as its */
/* second parameter the channel for output to the host.      */

process (stacksize = 10K, heapsize = 100K,
         interface (input host_in, output host_out, input secondary)
         ) controller;


/* The subordinate will reside on a node remote from the host.*/
/* It will not have access to the host's services, and so     */
/* should not have those connections defined */

process (stacksize = 2K, heapsize = 4K,
         interface (output result)
         ) subordinate;

/* The network has two edges; and these will be mapped onto   */
/* the host */

input HostInput;       /* Define the edge from the host */
output HostOutput;     /* Define the edge to the host   */

connect HostInput to controller.host_in;
connect HostOutput to controller.host_out;

connect controller.secondary to subordinate.result;

/* These can now be mapped - the controller on the root, and */
/* the subordinate on to the other one. */

place controller on root;

place subordinate on servant;

/* And the edge connections placed on the host connection    */
```

```
place HostInput on host;  /* say that the software edges are */
place HostOutput on host; /* mapped onto the hardware        */
                          /* host edge */

use "hello.lku" for controller;
use "world.lku" for subordinate;
```

Schematically the configuration is as follows:



Note: inter-process channels between `controller` and `subordinate` will automatically be placed by the configurer on the link between `root` and `servant`.

To complete the construction of the application, the programs must be configured and collected:

(UNIX)

```
icconf twin.cfs -o twin.cfb
icollect twin.cfb -o twin.btl
```

(MS-DOS and VAX/VMS)

```
icconf twin.cfs /o twin.cfb
icollect twin.cfb /o twin.btl
```

This produces the file `twin.btl`, which can be booted from the host and will print:

```
Hello World
```

6.2 Configuration language

This section describes various aspects of the configuration language and ends with a language summary. The syntax of the configuration language is given in appendix B.

6.2.1 Introduction

The network configuration language is a special purpose language that allows linked object code to be connected to other linked units and placed on any physical arrangement of transputers. The language has been designed to be compatible with INMOS toolsets and allows linked modules from these toolsets to be mixed on the same network.

The main features of the language are listed below.

- The language is declarative.
- Software and hardware networks are described using a common syntax.
- Identifiers have global scope (except replication counters).
- Arrays can be declared of any symbolic element, including processes, processors, channels, and edges.
- Replicative and conditional statements allow easy declaration of regular networks and exceptions within them.
- New node types can be defined.
- Source files can be included.
- Comments can be inserted at any point.

A formal description of the language can be found in appendix B. The following sections describe the main features of the language and explain each of the language statements.

6.2.2 Statements

All statements, except the conditional statement `if` and the replication statement `rep`, must be terminated by a semicolon. Blocks of statements must be enclosed in braces '{' and '}'.

Indentation may be used to indicate structure in conditional and replicative statements. However, indentation is not required by the syntax.

6.2.3 Comments

Comments can appear anywhere in the configuration text and may extend over any number of lines. Comments must be preceded by the character sequence '`/*`' and followed by the character sequence '`*/`'. For example:

```
process worker; /* declare s/w process "worker" */
```

Comments may not be nested.

6.2.4 Identifiers

Identifiers are symbolic names for configuration elements such as processors, processes, channels, and edges. Identifiers can be undefined or can be associated with a *type* (see below) in a typed statement. Identifiers may be followed by dimensions to create array types.

Character set

Identifiers can contain any letter, digit, or the underscore character; they must begin with a letter or underscore. All characters in the name are significant and characters are case-sensitive.

6.2.5 Types

Six base types are defined in the language:

`node` `input` `output` `edge` `connection` *numeric value*

Hardware and software networks are described as collections of interconnected nodes. Each node has a set of attributes defined by the node type. The node types `processor` (hardware) and `process` (software) are predefined from a defaults file that is read when the configurer is invoked on the configuration description file.

Numeric values can have the following types:

```
char  int  float  double
```

`char` represents the signed 8–bit integer value of a character's ASCII code. `int` is a signed 32-bit integer, `float` is a 32-bit IEEE 754 single length real, and `double` is a 64-bit IEEE 754 double length real. Types for constants are implied by the form of the numeric value.

6.2.6 Constants

Numeric and character constants can be defined using the `val` statement. The type of the constant will be deduced from the expression. For example:

```
val gridsize 4;      /* integer assumed */
val x_coord 2.0f;    /* single length real */
val y (x * 23.2e3)   /* expression */
```

Integers can be expressed in decimal, octal, or hexadecimal. The suffixes K and M can be used as fixed multipliers to indicate 'Kilo' (2^{10}) and 'Mega' (2^{20}) values. Single length floating point numbers must be suffixed with the letter F or f.

Character constants must be enclosed within single quotes and string constants within double quotes. Standard escape sequences can be used to specify control characters such as Tab and EOL ('end-of-line'). For example:

```
val c 'c';                 /* character constant */
val greeting "Hello\n"; /* string constant */
```

Note: Any string constant that is to be passed to a C program must be explicitly terminated by the null character escape sequence \0. This is because the configurer does not automatically terminate strings with \0.

Escape sequences are defined in appendix B.5.3.

Constant arrays can be defined by enclosing the sequence of values in braces. Multidimensional constant arrays are also allowed. For example:

```
val pow2 {1, 2, 4, 8, 16, 32, 64, 128};
val powers {{1, 1, 1}, {2, 4, 8}, {3, 6, 9}};
```

6.2.7 Booleans

The boolean constants TRUE and FALSE are predefined as integer constants with values one and zero respectively. In conditional statements any *non-zero* expression counts as TRUE.

6.2.8 Expressions and arithmetic

Expressions follow the syntax used in the C programming language. Operator precedence determines the order of evaluation, and brackets can be used to override the normal ordering.

Operators supported are as follows:

Unary: ",,"- + ~

Binary: + - * / % & | ^ && || << >> < <= > >= == !=

Ternary: ?:

All integer arithmetic is carried out to 32 bit precision.

Strings and arrays can be tested for equality in the same way as integer expressions by using the == and != operators.

6.2.9 Arrays

Arrays can be declared for any base type or user-defined node type. For all array declarations except constant arrays, the dimensions are specified after the array name using the square bracket convention for subscripts. Subscripts are numbered from zero and values are stored in row order.

For constant array declarations all elements must be of the same type. In multidimensional constant arrays the dimension sizes of all the subarrays must be the same.

Elements of constant arrays can be referenced by specifying the subscript either after the array name or after the array declaration. For example:

```
val y x[i];
val x {1, 2, 3}[i];
```

Arrays are commonly used to define the basic elements of the hardware and software networks. For example:

```
processor grid[4];
process slave[4];
```

6.2.10 Conditional statement

The `if...else` statement controls the execution of the statement that immediately follows it. The syntax of the statement is as follows:

`if` *exp statement* [`else` *statement*]

where: *exp* is any valid expression;

statement can be a single statement or a group of statements.

`if` is commonly used to exclude part of a network from a replicated declaration. For example:

```
T425(memory=1M) grid[4];
edge freelink[4];

rep i = 0 to 3
  {
  connect grid[i].link[2] to grid[i].link[3];
  if (i == 0)
    connect grid[i].link[1] to host;
  else
    connect grid[i].link[1] to freelink[i];
  }
```

`if` can also be used to conditionally place a process on a specific processor, for example, to place a process on a remote processor in a network only if it is equipped with enough memory:

```
if remote.memory >= 2M
  place master on remote;
else
  place master on root;
```

6.2.11 Replication

The `rep` statement replicates the following statement or group of statements. `rep` is a counted loop in which the control bounds can be integers or integer expressions.

`rep` has two syntactic forms in which the number of iterations can be specified by a range or by an initial value followed by a count:

`rep` *index* = *exp1* `to` *exp2* *statement*

`rep` *index* = *exp1* `for` *exp2* *statement*

For example:

```
rep i = 0 to 9
  { ...
  }

rep i = 0 for 10
  { ...
  }
```

If the range or count is zero the succeeding statement or group of statements is not executed.

Replication is commonly used to define regular networks such as grids, rings, and hyper-cubes and to place processes on them. It can be used for both hardware and software networks.

The following example connects four T425 transputers in a square array and places the same process on each. The processors are connected to their neighbors via links 2 and 3; links 0 and 1 of the processor are left unconnected:

```
T425(memory=1M) grid[4];

rep i = 0 to 3
  connect grid[i].link[2] to grid[i].link[3];

process slave[4]

rep i = 0 for 4
  place slave[i] on grid[i];
```

6.2.12 Built–in functions

The function `size` (*array*) is built–in. `size` returns the number of elements in an array. If the argument is not an array then `size` returns the value 1 (one).

6.2.13 Network definition

Software and hardware networks are defined using a common syntax based on the declaration of nodes and their connections.

Nodes

Nodes are a generic network type from which hardware and software nodes can be defined. Node types `processor` and `process` are predefined in the configurer startup file.

Nodes are associated with a number of *attributes*,the exact number and nature of which depends on the value given to the `element` attribute of a node i.e. `process` or `processor`. Nodes with element type `processor` include attributes such as `type` and `memory`, whereas nodes of type `process` have a set of runtime process attributes such as the process interface, priority of execution, memory requirements, and code segment ordering.

The software node type `process` and the hardware node type `processor`, although not part of the formal language syntax, are predefined in a configuration defaults file which is read each time the configurer is invoked. This means that they can be used as though they are part of the language. Definitions of predefined types can be found in section B.3.2.

Node attributes

Node attributes can be accessed in expressions using the dot convention. For example, they can be used to control the placing of processes:

```
if (remote.memory >= 2M)
  place p on remote;
else
  place p on root;
```

When assigning values to sub–attributes open and closed brackets '`()`' are used in place of the dot. For example, the statements:

```
process(order(code=...));
process.order(code=...);
process(code=...);
```

are equivalent. (Where *process* is the name of a declared process or process type).

Sub–attributes can be specified without specifying the parent attribute. Because all attributes are unique for node type `processor`, ambiguity is not a problem. This is not the case for `process` nodes. For example, the statements:

```
process(location(code=...));
process.location(code=...);
```

are equivalent, however, the statement:

```
process(code=...);
```

is the same as:

```
process(order(code=...));
```

The `order` and `location` attributes associated with the `process` node type have the same sub–attributes. In cases where the parent attribute (i.e. `order` or `location`) is not specified, the configurer will assume that an `order` attribute is intended.

Process and processor attributes are described in detail in sections 6.1.2 and 6.1.3 respectively.

Defining new node types

Refinements of existing node types can be created by using the `define` statement to specify nodes with specific attributes. As an example, consider the definitions of the node types `process` and `processor` which are in the configurer startup file:

```
define node(element="processor") processor;
define node(element="process") process;
```

Once defined, node types can be used to define other node types. For example, the `processor` type can be used to define a specific transputer type which can then be refined into a TRAM definition:

```
define processor(type = "T425") T425;
define T425(memory = 1M) B411;
```

New software node types can be defined in the same way. For example:

```
define process(stacksize = 10K,
               interface(int count,
                         input command,
                         output result)) workpackage;
```

Once defined, new types can be used to declare variables in the same way as base types. For example:

```
T425 worker;

B411 root;

workpackage slave[4];
```

Connections

Nodes are connected by the `connect` statement which can be used to join software channels (unidirectional), transputer links (bi-directional), or network edges (bi-directional for hardware and uni–directional for software). The statement has two syntactic forms:

`connect` *item* `to` *item* `[ by` *connection* `] ;`

`connect` *item*`,` *item* `[ by` *connection* `] ;`

Connections can also be named for later use in the configuration, using a name of the connection type.

Prohibited connections

The following connections are disallowed and generate a configurer error:

- Processes to processors
- Inputs to inputs (except channel to edge connections)
- Outputs to outputs (except channel to edge connections)
- Network edge to network edge

6.2.14 Configuration language summary

Network data types

`node`	A point in a software or hardware network. Has the general attribute `element` (defined as `process` for a software network and `processor` for a hardware network), and other attributes defined by the value given to `element`.
`connection`	A named connection between links or channels.
`edge`	Declares a network edge.
`input`	Declares a software process *input* channel or edge.
`output`	Declares a software process *output* channel or edge.

Numeric data types

`int`	Integer type.
`char`	Character type.
`float`	Single length floating point type (IEEE 754).
`double`	Double length floating point type (IEEE 754).

Language constructs

`if`	`if` *exp statement* [`else` *statement*] Simple conditional construct. *exp* can be any valid integer expression and *statement* can be the single succeeding statement or a group of statements. `else` *statement* is optional.
`rep`	`rep` *index* `=` *exp1* `to` *exp2 statement* `rep` *index* `=` *exp1* `for` *exp2 statement* Simple replication construct. Can be controlled by a range or a count.

`connect`	`connect` *item* `to` *item* `by` *connection*`;`
	`connect` *item*`,` *item* `by` *connection*`;`
	Joins channels to channels, links to links, channels to software edges, and links to hardware edges. `by` *connection* is optional.

`place`	`place` *process* `on` *processor*`;`
	`place` *channel* `on` *link*`;`
	Assigns a software process to a processor, a channel to a link, a software edge to a hardware edge or a named channel connection to a named link connection.

`use`	`use` *filename* `for` *process*`;` Assigns a linked unit to a process.
`#include`	`#include` *filename* Includes another source file.

Definitions

`val`	`val` *identifier exp*; Defines a numeric constant. The type is deduced from the type of the expression.
`define`	`define` *type* (*attributes*) *identifier;* Defines a node type. A list of attributes is optional.

Operators

Unary:	`+ - ","~`
Binary:	`+ - * / % & | ^ << >> && || < > <= >= == !=`
Ternary:	`?:`

Predefinitions

(Node types)

`process` `PROCESS`	Software process node type.
`lowprocess` `LOWPROCESS`	Software low priority process node type.
`highprocess` `HIGHPROCESS`	Software high priority process node type.
`processor` `PROCESSOR`	Hardware processor node type.

`T212 t212 T222 t222`
`T225 t225 M212 m212`
	IMS T2 processor series.

`T400 t400 T414 t414`
`T425 t425 T426 t426`
	IMS T4 processor series.

`T800 t800 T801 t801`
`T805 t805`
	IMS T8 processor series.

(Constants)

`HIGH, high`	The integer constant 0 (zero). Used to indicate a high priority process.
`LOW, low`	The integer constant 1 (one). Used to indicate a low priority process.
`TRUE, true`	The integer constant 1 (one).
`FALSE, false`	The integer constant 0 (zero).

`MIN_COST` `min_cost`	The integer constant 1. Used by the `routecost` processor attribute.
`MAX_COST` `max_cost`	The integer decimal constant 1000000. Used by the `routecost` processor attribute.
`DEFAULT_COST` `default_cost`	The integer decimal constant 1000. Used by the `routecost` processor attribute.
`INFINITE_COST` `infinite_cost`	The integer decimal constant 1000001. Used by the `routecost` processor attribute.
`ZERO_TOLERANCE` `zero_tolerance`	The integer constant 0 (zero). Used by the `tolerance` processor attribute.
`DEFAULT_TOLERANCE` `default_tolerance`	The integer constant 1. Used by the `tolerance` processor attribute.
`MAX_TOLERANCE` `max_tolerance`	The integer decimal constant 1000000. Used by the `tolerance` processor attribute.
`ROUTER_ORDER` `router_order`	The integer decimal constant –20000. Weights the relative position of virtual routing system processes in memory.
`MUXER_ORDER` `muxer_order`	The integer decimal constant –10000. Weights the relative position of software multiplexing system processes in memory.

Note: use of this second block of constants is described in chapter 9 – '*Advanced use of the configurer*'.

(Edges)

`host`	The host link or channel.

Built–in functions

`size`	Returns the size of an array.

6.3 Further considerations

Now that the basic structure of the configuration process and language has been described, some further considerations need to be addressed. This section discusses the following topics:

- Runtime library
- Reliable channel communication
- Terminating configured processes
- Checking the configuration
- Debugging configured programs with `idebug`.

6.3.1 Runtime library

As mentioned above, in section 6.1.4, the channels are mapped automatically onto the available hardware links. If channels are between processes on adjacent processors, then they can be placed directly onto the link that joins the processors. If there are no other channel connections that wish to use that link connection, then the channel is a *direct* channel. The channel can also be termed direct if it connects processes that are on the *same* processor (i.e. a soft channel).

If more than one channel pair connection is to use a link, then all those channels are *virtual*.

There is a special case when the interactive debugger `idebug` is in use. In this case, it should be assumed that *all* channels that connect processes on different processors are virtual, because the debugger will insert its own routing requirements on top of those for the application.

The library functions for channel communication, `ChanIn`, `ChanInInt`, `ChanInChar`, `ChanOut`, `ChanOutInt`, `ChanOutChar` can all be used on any type of channel. They will adapt to whether the communication is direct or virtual.

However, the functions whose names begin '`DirectChan...`' must *not* be used on virtual channels. They will only work for direct or soft channels. Typically, they should be reserved for the two cases:

> Communication between processes in the *same* C program, i.e. the same linked unit. In this case the channel is known to be a soft channel.
>
> Or:
>
> Communication on a link to a device outside the network, i.e. on an edge that is not the host connection.

It is important to remember that any process could be incorporated by another configuration in a different way i.e. when software is run on different hardware or when different configuration options are specified. The manner in which a process uses a channel is part of its specification. It may not be appropriate to assume that it will always be implemented as a direct or soft channel.

6.3.2 Reliable Channel Communications

There are a number of library functions that can be used to handle faults in the communication network. These can be used only on *direct* channels. They must not be used on virtual channels, nor during debugging.

The functions are:

```
int ChanInTimeFail (Channel *chan, void *cp, int cnt,
                    int time)

int ChanOutTimeFail (Channel *chan, void *cp, int cnt,
                     int time)

int ChanInChanFail (Channel *chan, void *cp, int cnt,
                    Channel * failchan)

int ChanOutChanFail (Channel *chan, void *cp, int cnt,
                     Channel * failchan)
```

The functions attempt a transfer of data on a channel. They return zero if the communication succeeded normally, and they return one if the communication was aborted.

The first three parameters are similar to the normal versions of these functions. The last parameter defines either the time to wait for completion of the communication, or a channel which will be used to terminate the communication if it has not succeeded.

These functions are *not* intended as the normal mode of communications. They have a higher overhead than other methods.

A further function is available to reset a channel that has gone awry in its communications. This is:

```
int ChanReset (Channel * chan)
```

When a hard link is quiescent, then it can be reset by this function.

Important note: These functions should not be used for checking the communications within a network if there is any doubt as to whether the data might not have transferred in a given amount of time. In general, you should be absolutely sure that the failure is due to a hardware failure, and not to the receiving or sending device being very busy. If the communication is terminated while data is actually being transmitted, then the results are undefined, and could stop one or both of the processors.

There is no point in using these functions on soft channels, because the communication in that case can be assumed to be secure.

6.3.3 Terminating configured processes

Configured processes (processes that have been configured on a processor by `icconf`) should use `exit` to terminate the program and iserver.

When it is required to terminate the program without terminating the server, the function `exit_noterminate` should be used. In addition the program must be linked with the full runtime system, by using `cstartup.lnk`.

> **Note:** The behavior of `exit` has changed since previous issues of the toolset e.g. the D7214, D6214, D5214 and D4214 products. In these products using `exit` in the configured case would not have terminated the server. `exit_terminate` is retained for compatibility with earlier issues, it has the same action as `exit`.

Details of the functions can be found in chapter 2 of the *ANSI C Language and Libraries Reference Manual.*

Configured processes which use the reduced library cannot terminate the server (even by using `exit_terminate`) because they have no link with the server.

6.3.4 Checking the configuration

Configurations may be checked against the hardware on a transputer board using a network check program such as `ispy`. The `ispy` program is supplied as part of the support software for some INMOS *iq* systems products. These products are available separately from your local INMOS distributor.

6.3.5 The effect of `icconf` on `idebug`

The use of `icconf` has a direct effect on the way in which the interactive/post mortem debugger `idebug` can be used to debug the program.

There are three main ways of using `icconf`:

- No special command line options, (the default) - this is not compatible with the debugger.
- With the `g` command line option - this is compatible with either the interactive or postmortem debugger. However, the real time performance of the bootable produced may be significantly different to that produced by default, if there is a high incidence of channel communication between processors.
- With the `gp` command line option - this is compatible with the postmortem debugger only. The real time performance of the bootable produced will only be slightly different to that produced by default.

Important note: when virtual routing processes are used, `idebug` cannot jump down channels between adjacent processors. If this is required, the configurer '`NV`' option should be used to disable virtual routing.

Table 6.1 summarizes the use of the relevant options.

`icconf` command options	Effect
'`g`' and '`nv`'	Interactive and post–mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`g`'	Interactive and post–mortem debugging enabled. Virtual routing enabled and forcibly used. *Not* possible to jump down channels between adjacent processors.
'`gp`' and '`nv`'	Post–mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`gp`'	Post–mortem debugging enabled. Virtual routing enabled and may be used. May be possible to jump down channels between adjacent processors.

Table 6.1 Effect of `icconf` options on debugging

6.4 Configuration examples

The examples presented here are intended to illustrate the syntax of configuration language statements and how they are used to form a configuration description. They are not intended as tutorial examples although examples 3 and 4 are provided in the toolset `examples/router` directory.

Further examples illustrating how to use the configuration language and configure software on various network topologies can be found in the `icconf` examples subdirectory. This subdirectory contains the program configuration source files and a number of Makefiles and batch files to assist with program building. A 'read me' file provides a summary of the directory contents, describes the prerequisite hardware, and gives instructions for building the programs.

Knowledge of the way the configuration language defines software and hardware networks and links them by mapping statements is a prerequisite to understanding the configuration model. Readers are recommended to study the examples at length and be thoroughly familiar with the language before attempting to write complex configurations.

6.4.1 Example 1 – single processor configuration

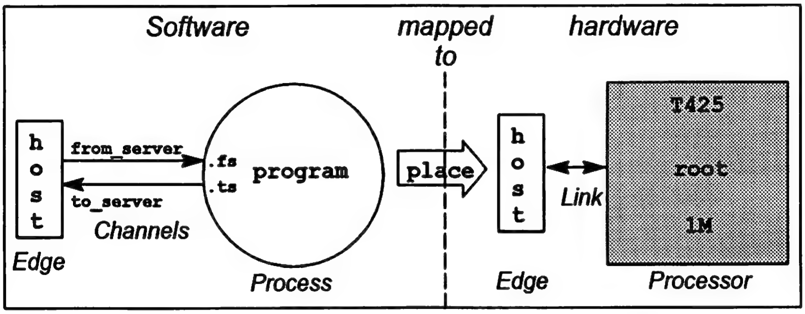
This example is almost identical to the example given in chapter 4, it is included here for completeness. The example configures the 'Hello World' program for a single T425 processor. Chapter 4 describes how to build and run a single processor program.

The configuration description first defines hardware data such as the processor type and network topology. In this case all that is required is to define the memory size of the processor and connect a link to the host edge.

Next it defines the software topology. Processes are named, channels are defined as input and output interface parameters, and connected as required. In this case the process channels are simply connected to server channels. Stack and heap sizes must be defined as parameters to the process.

Finally, the configuration names the linked file to use for each process and defines the mapping of processes to processors, and channels to links or edges.

Schematically the configuration is as follows:



The configuration description is as follows:

```
/* Hardware description:
   declare processor memory size;
   connect link 0 to host edge        */

T425 (memory = 1M) root;

connect host to root.link[0]; /* Link 0 connected */
                              /* to iserver */

/* Software description: declare channels;
   declare process and interface params;
   connect interface to inputs and outputs  */

input from_server;
output to_server;

process (stacksize = 100K, heapsize = 100K,
         interface (input fs, output ts)) program;

/* connect server channels to program */
connect from_server to program.fs;
connect to_server to program.ts;

/* Mapping description:
   define object file;
   place process on processor;
   place channels on predefined edge host */

use "hello.lku" for program;

place program on root;

place from_server on host;
place to_server on host;
```

6.4.2 Example 2 – Two processes configured on a two-processor network.

The program consists of two processes. One process acts as the interface to the host (the i/o process) and the other performs a complex numerical calculation.

The hardware consists of a T425 and a T800 transputer connected together by a single link. The T425 acts as the root transputer. The i/o process is to be executed on the T425 and the numerical process on the T800.

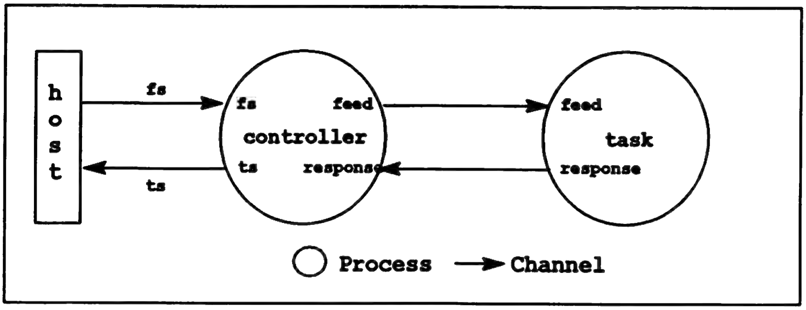


Figure 6.2 Example two process program with four channels

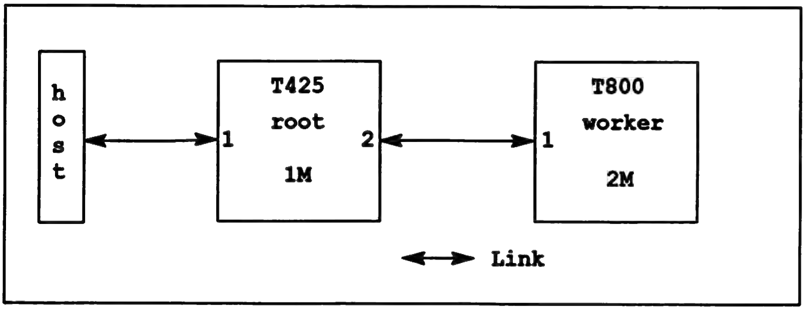


Figure 6.3 Example two processor network, two links connected

The configuration description is listed below:

```
/* Hardware description: */

T425 (memory = 1M) root;
T800 (memory = 2M) worker;

connect root.link[1] to host;

/* Software description: */

input fs;   /* input edge */
output ts;    /* output edge */

process (stacksize = 8K, heapsize = 50K,
         interface (input fs,
                    output ts,
                    output feed,
                    input response)) controller;

process (stacksize = 16K, heapsize = 512K,
         interface (input feed,
                    output response)) task;

connect fs to controller.fs;
connect ts to controller.ts;
connect controller.feed to task.feed;
connect controller.response to task.response;

/* Mapping description: */

use "control.lku" for controller;
use "compute.lku" for task;

place controller on root;
place task on worker;

place fs on host;
place ts on host;
```

6.4.3 Example 3 – using virtual channels

The program consists of two processes `Proc1` and `Proc2` with two eight–channel wide sets of connections between them. `Proc1` also acts as the interface to the host and requires access to the host link supported by `iserver`.

The hardware consists of an array of two T800 processors (`PROC[0]` and `PROC[1]`) each with 2 Mbytes of memory. The two processors are connected together by three links. `PROC[1]` is also attached to an edge by its fourth link.

Figures 6.4 and 6.5 describe the software and hardware networks respectively, the source for the example is supplied in the `examples/router` subdirectory.

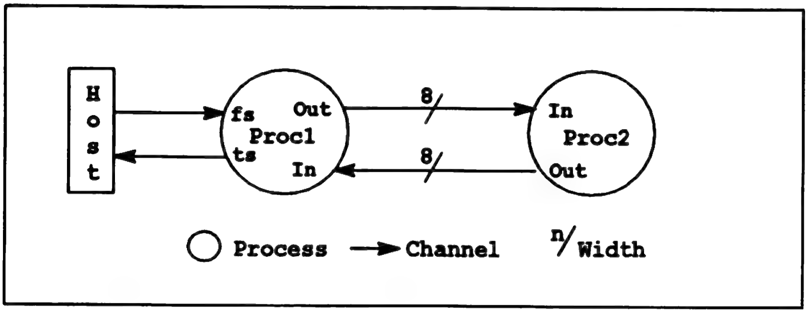


Figure 6.4 Example two process program with many channels

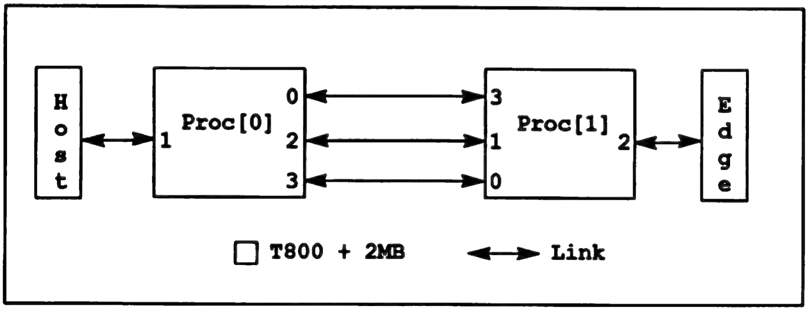


Figure 6.5 Example fully connected two processor network

As can be seen there are not enough links for each link to be dedicated to a single pair of opposing channels. It is therefore necessary for the configurer to share the links between the application channels using multiplexing and demultiplexing software.

The configuration description is as follows:

```
/* Hardware description for two B404-3 module subsystem */

define T800 (memory = 2M) B404_3;

B404_3 PROC[2];

edge pipe_down;

connect host              to PROC[0].link[1];
connect PROC[0].link[2] to PROC[1].link[1];
connect PROC[1].link[2] to pipe_down;

connect PROC[0].link[0] to PROC[1].link[3];
connect PROC[1].link[0] to PROC[0].link[3];

/* Software description for highly connected two processes */

val BusWidth 8;

input fs;
output ts;

process (stacksize = 2k, heapsize = 16k,
         interface(input fs, output ts,
                   input In[BusWidth],
                   output Out[BusWidth])) Proc1;

process (stacksize = 2k, heapsize = 16k,
         interface(input In[BusWidth],
                   output Out[BusWidth])) Proc2;

rep i = 0 for BusWidth
{
  connect Proc1.Out[i] to Proc2.In[i];
  connect Proc2.Out[i] to Proc1.In[i];
}

connect Proc1.fs to fs;
connect Proc1.ts to ts;

/* Mapping description */

place Proc1 on PROC[0];
place Proc2 on PROC[1];

use "proc1.lku" for Proc1;
use "proc2.lku" for Proc2;

place fs on host;
place ts on host;
```

6.4.4 Example 4 – Virtual channel routing:

The program is the same as described in figure 6.4, however, it is loaded onto two non-adjacent processors in a three processor network (see figure 6.6). In this case no link directly connects the processors used. This example shows how `icconf` can utilize otherwise unused processors in the target network for through–routing channels of other processors. The source for the example is supplied in the `examples/router` subdirectory.

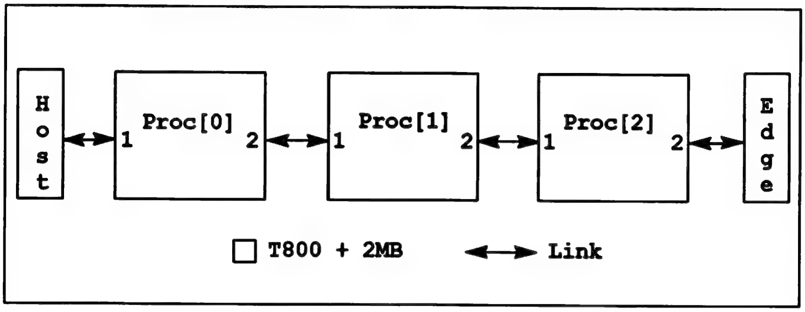


Figure 6.6 Example sparsely connected three processor network

For this example a new hardware description is required but the software description is the same as that used in the previous example. The mapping description also requires very little modification. The configuration description is as follows:

```
/* Hardware description for three B404-3 module subsystem */

define T800 (memory = 2M) B404_3;

B404_3 PROC[3];

edge pipe_down;

connect host            to PROC[0].link[1];
connect PROC[0].link[2] to PROC[1].link[1];
connect PROC[1].link[2] to PROC[2].link[1];
connect PROC[2].link[2] to pipe_down;

/* Software description for highly connected two processes */

val BusWidth 8;

input fs;
output ts;

process (stacksize = 2k, heapsize = 16k,
         interface(input fs, output ts,
                   input In[BusWidth],
                   output Out[BusWidth])) Proc1;

process (stacksize = 2k, heapsize = 16k,
         interface(input In[BusWidth],
                   output Out[BusWidth])) Proc2;

rep i = 0 for BusWidth
{
  connect Proc1.Out[i] to Proc2.In[i];
  connect Proc2.Out[i] to Proc1.In[i];
}

connect Proc1.fs to fs;
connect Proc1.ts to ts;

/* Mapping description */

place Proc1 on PROC[0];
/* No application software on PROC[1] */
place Proc2 on PROC[2];

use "proc1.lku" for Proc1;
use "proc2.lku" for Proc2;

place fs on host;
place ts on host;
```

7 Loading transputer programs

This chapter explains how to load programs onto single transputers and transputer networks. It briefly describes the format of loadable programs and introduces the program loading tools `iserver` and `iskip`. The chapter goes on to explain how to load programs for debugging and ends with an example of skip loading.

7.1 Introduction

Transputer programs are loaded onto transputer boards with the `iserver` tool which installs code on each processor using processor and distribution information embedded in the executable file. The executable file consists of code to which bootstrap information has been added to make the program self-booting on the transputer. Self-booting executable code is also known as *bootable* code.

Bootable files are generated by `icollect` from configuration data files (network programs) or linked units (single transputer programs). Bootable files are generated with the default extension `.btl` (for loading onto boot from link boards), or `.btr` (for loading onto boot-from-ROM boards). **Note**: a bootable file is constructed such that copying it to a link will boot the network automatically.

7.2 Tools for loading

Two tools are provided to load programs onto transputers and transputer networks:

- `iserver` — the file server and loader tool.

  `iserver` loads the bootable file onto the single transputer or transputer network and activates the host file server that provides communication with the host.

- `iskip` — the skip loading tool.

  `iskip` allows a program to be loaded over the root transputer onto an external network. The tool is used prior to invoking `iserver` to start up a special route-through process on the root transputer that transfers data between the the network and the host system.

Skip loading is useful for the post-mortem debugging of programs that do not use the root transputer. The root transputer in the network is omitted from the logical network and the program is loaded onto the first processor *after* the root transputer, leaving it free to run the debugger. This avoids having to debug the code from a memory dump file.

Programs loaded using `iskip` always require one extra processor on the network in addition to those required to run the program. For example, a program written for a single transputer requires at least two processors, one to act as the root transputer and one to run the program.

7.3 The boot from link loading mechanism

`iserver` loads programs onto transputer networks, via the host link connection. It does this by simply copying the contents of the bootable file to the link. The bootable file contains all the bootstrap and loader code to ensure that the program is loaded onto the network and starts running.

The server has to be told which link connection to use and how to access it. This is done by specifying the name of a User Link on the command line or in the environment variable `TRANSPUTER`. The server gets information about the specified User Link from a connection database file. See the `iserver` documentation in chapter 13 of the *Toolset Reference Manual.*

The bootstrap code for the transputers in the network is sent first. The code is propagated through the network as individual processors load neighboring processors. After all of the transputers in the network have been booted, program code is loaded onto individual processors. For a multitransputer network the allocation of processes to processors is determined by the configuration file. For single transputer programs code is loaded onto the first processor on the network.

When all of the code is loaded into the transputer's memory, the program starts running and can communicate with the host using the standard library routines for input and output. The libraries actually communicate with the host via the server using a predefined communication protocol known as the 'SP' protocol. This protocol is defined in the `iserver` documentation.

The program continues to run until: an error occurs, the server is terminated by pressing the `iserver` interrupt key (usually CTRL-C or CTRL-BREAK), or the program terminates naturally. Note: terminating the server will not stop the program running on the transputer. However, any processes on the transputer which attempt to communicate with the server will deadlock. This may eventually cause the whole program to stop as other processes become dependent on this communication. The program may be able to continue if the server is restarted.

If `iskip` is used, the first transputer in the network is bypassed. Therefore the network must contain one additional transputer to the number required to run the program.

7.4 Boards and subnetworks

There are two basic types of transputer evaluation board: those that boot from link and those that boot from ROM.

Boot from link TRAM boards form the majority of transputer boards in general use. They are loaded down the link that connects the root transputer to the host using the `iserver` tool. Programs intended to run on boot from link boards must consist of bootable code, such as that generated by `icollect`.

Examples of boot from link boards supplied by INMOS are the IMS B008 PC motherboard (with appropriate TRAMs) and the IMS B014and IMS B016 VME bus standard interface boards.

Boot from ROM TRAM boards are intended for stand–alone applications such as embedded systems.

7.4.1 Subsystem wiring

Subsystem wiring is the way in which boards are connected together, and determines the manner in which transputer subnetworks are controlled.

Three signals are used to control transputers mounted in a system, namely **Reset**, **Analyse**, and **Error**. Together these are known as the *system services*. All INMOS transputer boards use a common scheme for propagating these signals to other subnetworks. The scheme is as follows.

Each transputer board has three ports for communicating system services from one board to another. These are *Up*, *Down*, and *Subsystem*. Up is the *input* port, used to control the board from an external source; Down and Subsystem are both output ports and are used to propagate the Up signals to other boards or subnetworks.

The Down and Subsystem ports work in the following ways:

Down propagates the Up signal unchanged to the next board or subnetwork. This allows multiple boards to be chained together by connecting successive Up and Down ports and the whole network can be controlled by a single signal.

Subsystem propagates the **Reset** and **Analyse** signals but also allows control by the board, enabling subnetworks downstream of the board to be independently reset, analyzed, and their error flags read, under the control of the transputer to which the subsystem is attached.

7.4.2 Connecting subnetworks

Multiple transputer systems can either be controlled by the host computer or by a *master* transputer controlled by the host computer.

In a typical multitransputer system the root transputer's Up port is connected to the host computer so that the host can control the loading of programs and monitor errors on the network. The first processor in the subnetwork is connected to either Down or Subsystem depending on the application, and other processors on the network are chained together via their Up and Down ports.

In a simple application requiring multiple transputers, the subnetwork would normally be connected to Down on the root transputer. This would allow the host computer to reset the whole network in a single operation and to monitor the error signal on any transputer in the network.

A more complicated application may require several programs to be loaded onto the subnetwork under the control of the root transputer. Here the subnetwork would be connected to Subsystem so that the root transputer could repeatedly reset and re-load the subnetwork. Any errors in the subnetwork would be detected by the root transputer through its Subsystem port, and the error would not be propagated through the Up port to the host computer. **Reset** and **Analyse** signals are propagated through to the Subsystem port, but the error signal is not relayed back. (Note some boards do not conform to this system of signal propagation – see section 7.5.2).

7.5 Loading programs for debugging

Special debugger and server options must be used for the debugging of programs running on transputer boards. The options vary with the subsystem wiring, the board type, and whether or not the program uses the root transputer. The effects of subsystem wiring are described above; the effects of board type and program mode are described in the following sections.

Commands to use for various combinations of subsystem wiring, board type, and program mode, are listed in the debugger reference documentation.

7.5.1 Breakpoint debugging

Programs are loaded for breakpoint debugging using the `idebug` command. When invoked in interactive mode this command incorporates a skip load and `iserver` is not required. Because it uses a skip load, breakpoint debugging requires at least two processors on the network.

7.5.2 Board types

Some early INMOS boards of the B004 type, unlike later TRAM-based boards, do not propagate Reset through to the Subsystem port. On these boards the '`A`' debugger option must be supplied on the debugger command line to reset the network.

7.5.3 Use of the root transputer

The use made of the root transputer by the program changes the procedures you must use in post-mortem debugging. This is because the debugger program executes on the root transputer and any application code becomes overwritten when the tool is invoked. Two procedures can be used to load and debug code running on the root transputer:

1 Programs can be loaded in the normal way using `iserver`, and the program image in the root transputer's memory saved to a file. The code running on the root transputer is then debugged from the dump file. Code running on the rest of the network is debugged in the normal way by reading the transputer memory directly down the transputer links. The dump file is created by invoking `idump`. The debugger is subsequently invoked using the debugger '`R`' option that directs it to read the dump file.

   Note: On boards that contain only one transputer this method *must* be used.

2 Programs can be loaded over the top of the root transputer by invoking the `iskip` tool before running `iserver`. This leaves the root transputer free to run the debugger. The program can then be debugged down the root transputer link in the normal way.

   If `iskip` is used an extra processor is required over and above those required to run the application program.

Programs configured for a subnetwork that does not include the root transputer can be loaded with `iskip` and `iserver` and debugged down the root transputer link using the debugger '`T`' option.

Details of the procedures to use for loading and debugging all types of transputer programs can be found in the debugger documentation.

7.5.4 Analyse and Reset

Care must be taken that **Analyse** or **Reset** are only asserted once on a network that is to be debugged, or incorrect data will be obtained. To ensure this the debugger should be invoked using the standard command sequences given in the debugger reference documentation.

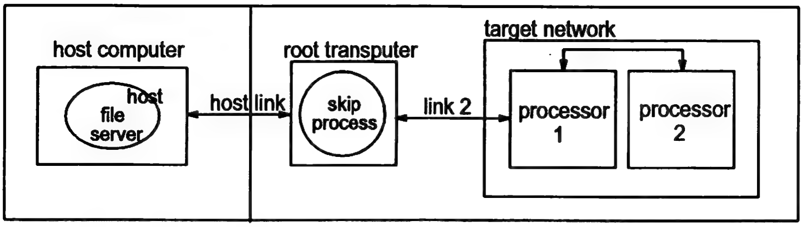
7.6 Example skip load

This section shows how to load a program into a network over the root transputer using the `iskip` tool.

7.6.1 Target network

The program to be loaded is configured for a target network consisting of two T800 processors mounted on a B008 motherboard. A T414 processor in slot zero acts as the root transputer, and the target network is connected to link 2 on the root transputer via one of the links on processor 1. The two T800 processors are connected by a single link.

The target network and its connections are shown schematically below.



7.6.2 Loading the program

The file **twinprog.btl** contains the bootable program.

To prepare the board for running the program on the target network, invoke **iskip** using one of the following commands:

```
iskip 2 -r -e                     (UNIX)
iskip 2 /r /e                     (MS-DOS and VMS)
```

This sets up the system to direct the program to the target network over the top of the root transputer and starts the route-through process on the root transputer. Options '**R**' and '**E**' respectively reset the target network and direct the host file server to monitor the halt-on-error flag. The program can then be loaded using one of the following commands:

```
iserver -ss -se -sc twinprog.btl     (UNIX)
iserver /ss /se /sc twinprog.btl     (MS-DOS and VMS)
```

Note: these examples assume that the environment variable **TRANSPUTER** has been defined to specify the name of the User Link to use to access the transputer network, and that a connection database file exists to define that User Link. See the **iserver** documentation (chapter 13 of the *Toolset Reference Manual*) for more detail.

See chapter 15 of the *Toolset Reference Manual* for more information on the **iskip** tool.

7.6.3 Clearing the network

On transputer boards error flags can be cleared using a network check program such as **ispy**. (Error flags can become set when the board is powered up).

The **ispy** program is provided as part of the support software for some INMOS *iq* systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program to clear the network is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared.

8 Debugging transputer programs

This chapter describes how to debug transputer programs. It describes the facilities of the toolset debugger `idebug` and shows how they can be used to debug transputer programs in a systematic manner. It explains how the debugger can be used in two modes (post-mortem and interactive) to analyze transputer programs and describes the two debugging environments (source code symbolic and low level monitor page). The chapter ends with some hints about debugging transputer programs and a list of points to note when using the debugger.

Worked examples are given at the end of this chapter.

Chapter 4 of the *Toolset Reference Manual* provides detailed information about `idebug`, including command line syntax, and full descriptions of the symbolic debugging and monitor page commands.

8.1 Introduction

The network debugger `idebug` is a symbolic debugger for transputers and transputer networks. It can be used to examine stopped programs (post-mortem debugging) or to debug programs interactively (breakpoint debugging). It can be used with INMOS FORTRAN-77, ANSI C, and occam programs, and with mixed language systems.

Programs can be analyzed using the *symbolic* functions which operate using source code symbols or the *monitor page* commands which operate at memory and processor level.

Symbolic functions allow files to be examined, variables inspected, and procedures traced, from source code level. Monitor page commands allow transputer memory to be examined and processor state to be determined anywhere on the network. Symbolic and monitor page environments are separate but can be switched between at will.

`idebug` can be used to debug mixed language programs, although certain facilities are available for some languages and not others. For example, a comprehensive expression language exists for C but not for occam. The exact usage of some of the facilities may also differ slightly between languages.

8.1.1 Post-mortem debugging

Post-mortem mode debugging allows stopped programs to be analyzed from the residual contents of transputer memory or from a network dump file. Programs that run on the root transputer must be debugged from a memory dump file because the debugger overwrites the root transputer's memory. The memory dump file is created using the `idump` tool (see chapters 4 and 5 of the *Toolset Reference Manual*).

8.1.2 Interactive debugging

Interactive debugging (also known as *breakpoint mode* debugging) allows transputer programs to be executed interactively using breakpoints set in the code.

Breakpoints can be set symbolically on lines of source text or at transputer memory addresses, and values can be modified in transputer memory to show the effect of changing variables. Breakpoint mode debugging requires the use of two or more transputers, because the debugger tool runs on the root transputer.

Certain symbolic functions and monitor page commands are only available in breakpoint debugging mode.

8.1.3 Mixed language debugging

When debugging programs constructed from a mixture of languages from different INMOS toolsets, you should always use the version of `idebug` with the highest version number (as displayed in the help or monitor pages). This is true for all versions of `idebug` with a version number greater than `V2.00.00`. This will help ensure that no toolset incompatibilities occur.

8.1.4 Debugging with `isim`

The transputer simulator tool `isim` can also be used to debug transputer programs from a low level environment. Using a similar environment to the debugger monitor page transputer memory can be examined, breakpoints set, and programs executed by single stepping.

The debugging facilities of the simulator are briefly described in this chapter (section 8.13). Details of how to use the simulator tool can be found in chapter 14 of the *Toolset Reference Manual*.

8.2 Programs that can be debugged

The debugger can analyze programs running on transputers that are either directly attached to a host through a server program, or connected to the host via a root transputer.

The *root* transputer is the processor that is directly connected to the host computer. In a transputer network that is connected to the host it forms the root of the network. The debugger always runs on the root transputer, which must be a 32-bit transputer with at least one megabyte of memory (preferably two or more).

The relationship of the root transputer to the host computer and the rest of the network is illustrated in figure 8.1.

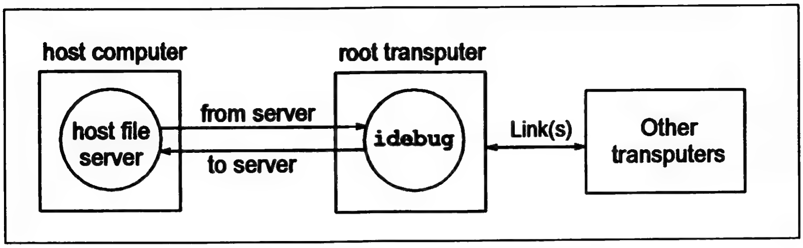


Figure 8.1 Debugging a transputer network

If breakpoint debugging is used the transputer network must contain at least two processors because the root transputer is dedicated to running the breakpoint debugger in parallel with the user's program.

8.3 Compiling programs for debugging

Programs to be debugged should be compiled with full symbolic debugging information enabled. For C and FORTRAN, this is achieved by specifying the compiler '`G`' option when the program is compiled. The occam compiler generates object files containing full debugging information by default. Two command line options may be used to limit the debugging information produced by the compiler.

Minimal debugging information

By default the C and FORTRAN compilers generate object files containing minimal symbolic debug information so that object modules, especially those to be used as libraries, are kept as small as possible. Minimal debug information enables the debugger to backtrace out of a library function to a module compiled with full debug information.

occam programs can be compiled with minimal debug information by using the compiler '`D`' option.

Note: The object code produced by the C and FORTRAN compilers with minimal debug information contains certain optimizations that are absent in code generated with full debugging information enabled. As a consequence the object code produced may differ slightly from code compiled with full debugging information enabled.

occam channel communication

The '`Y`' option to the occam compiler disables channel communication via library routines and, instead, produces optimal in-line code for channel i/o. Interactive debugging requires all communications to be done by means of the library routines, so this option also disables interactive debugging.

C channel communication

Use of the C library `DirectChan` functions on channels provided by the configurer will interfere with and corrupt interactive debugging. Note that the `DirectChan` functions can be safely used with edges passed from the configurer, and with internal (soft) channels declared in C source files.

8.3.1 Error modes

Programs to be debugged should be generated in HALT mode, which is the linker default. The behavior of a program when an error occurs depends on the mode in which the program was compiled and linked, as follows:

- In HALT mode any error during program execution halts the transputer immediately.
- In STOP mode, errors do not halt the program, rather they stop the errant process allowing other processes executing on the same transputer to continue. Programs compiled in this mode can only be debugged if they are halted explicitly.
- Programs compiled in UNIVERSAL mode will adopt the error mode selected at link time i.e. HALT or STOP mode. If UNIVERSAL mode is selected at both compile and link time, then the error behavior will default to HALT mode.

By default, C and FORTRAN programs are compiled in UNIVERSAL error mode and linked in HALT mode. By default, occam programs are compiled in HALT mode and linked in HALT mode.

8.4 Debugging configured programs

Programs configured with the C-style configurer, `icconf`, must have debugging enabled by means of the appropriate `icconf` command line options. Table 8.1 summarizes the use of the relevant options.

In mixed language programs incorporating occam modules, the occam code should be compiled and linked with interactive debugging enabled (the default).

8.4.1 Debugging with configuration level channels

`idebug` cannot locate to a process waiting on a transputer link, or locate to a process (on a different processor) waiting on a channel mapped onto a link, if that link is used by the configurer for software virtual channels.

`idebug` *is* able to locate to a process waiting on a transputer link or jump down a channel between two processes (which may be on different processors) if the channel is one of the following:

- An internal (soft) channel between processes on the same processor.
- An external (hard) channel between processes on different processors which is not used by the configurer for software virtual links.

8.4.2 Debugging with the configurer reserved attribute

The reserved attribute should *not* be specified to the configurer in order to reserve on-chip memory if you wish to interactively breakpoint debug the program. This is because the runtime kernel (see section 8.7.1) which the debugger places on each processor reserves the first 11K – 15K of memory for its own use (regardless of the `reserved` attribute being specified to the configurer).

8.5 Debugging boot from ROM programs

Programs configured using the configurer '`GP`' option (see table 8.1) may also be debugged in boot from ROM run in RAM systems (configurer '`RA`' option).

`icconf` command options	Effect
'`g`' and '`nv`'	Interactive and post–mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`g`'	Interactive and post–mortem debugging enabled. Virtual routing enabled and *will* be used. *Not* possible to jump down channels between adjacent processors.
'`gp`' and '`nv`'	Post–mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`gp`'	Post–mortem debugging enabled. Virtual routing enabled and *may* be used. Only possible to jump down channels between adjacent processors if they are *not* used for virtual routing.

Table 8.1 Effect of `icconf` options on debugging

8.6 Post-mortem debugging

Post-mortem debugging is the analysis of stopped programs, that is, programs that have failed to run correctly and have set the transputer error flag (or have detected a *hard* parity error). Programs that are to be debugged in this mode should be compiled and linked in HALT mode (HALT is the linker default) so that the processor halts when the flag is set. They should be loaded by **iserver** using the 'SE' option, so that the error flag is monitored and the server terminated if the error flag is set.

The conditions in which the transputer error flag may be set depend on the language being used. C and FORTRAN provide little or no automatic checking of errors whereas occam provides comprehensive error checking by default.

C programs can also set the error flag and halt the processor when the program is terminated by functions such as **halt_processor**, **abort**, **assert**, **debug_stop** or **debug_assert**.

8.6.1 C and FORTRAN programs

Little automatic error checking is provided in C or FORTRAN — this can make it difficult to cause a program to halt when an error occurs. This rather restricts the usefulness of post-mortem debugging, but it can be used if programs are halted explicitly using the debugging support functions such **debug_assert()** (**DEBUG_ASSERT()** in FORTRAN) etc. These functions are described more fully in the appropriate *Language and libraries* manual and in the debugging examples.

Breakpoint debugging, with its associated debugging support functions, is a more flexible approach and is the recommended method when debugging C and FORTRAN programs.

The C library **abort()** function can be enabled to halt the processor by calling the auxiliary function **set_abort_action()**. This enables a backtrace to be performed to the point in a program where the error occurred without the need to modify any of the **assert()** statements contained in the program.

This technique is illustrated with the following example (which is contained in the C toolset debugger examples directory):

```
/*****************************************
*
*  Debugger example:  abort.c
*
*  Example of forcing a C program to HALT the
*  processor for post-mortem analysis regardless
*  of the error mode it has been configured in.
*
*  Use of the debug support functions is encouraged
*  as an alternative (see debugger example file debug.c
*  for further details).
*
*****************************************/

#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <assert.h>

int
main (void)

{
        /* 0 will cause assert() to fail assertion test */
        int     x = 0;

        printf ("Program started\n");

        /*  override normal abort action  */
        set_abort_action (ABORT_HALT);

        printf ("Program being halted by assert ()\n");
        assert (x);

        printf ("Program being halted by abort ()\n");
        abort ();

        exit (EXIT_SUCCESS);
}
```

8.6.2 occam programs

The runtime errors that can cause an occam program to set the error flag and halt include:

- An arithmetic error, such as overflow or divide by zero, has occurred.
- An array index is out of range.
- A value is out of range in a type conversion.
- An alignment error has occurred in a type conversion or abbreviation
- An array element is being 'aliased' at run-time — that is, being referred to by more than one name within a given scope.
- A `STOP` process, or a process which behaves like `STOP` (e.g. an `IF` with no `TRUE` guards, or an `ALT` with no enabled guards), being executed.

When a run-time error occurs, the program halts the processor and allows the debugger to enter the program for post-mortem debugging.

In addition, some debug support functions (e.g. `DEBUG.ASSERT()`) are provided to aid debugging of programs by implementing an explicit program error; details of these functions can be found in the *occam language and libraries* manual and in the debugging examples.

8.6.3 Interrupted programs

Post-mortem debugging can also be used to debug programs that have been explicitly interrupted with the host system BREAK key. To interrupt a program, for example when a program 'hangs', press the BREAK key, which stops the server but not the program, and then run `idump` to take a snapshot of the running program. Running `idump` stops the program by sending an **Analyse** signal to the transputer in order to take a snapshot of its current activity.

8.6.4 Parity errors

The T426 will detect two types of parity errors, hard and soft. A soft error is one which disappears on retry; it does not stop the processor but sets, and resets, the **SoftParityError** pin. This allows soft errors to be monitored externally (or internally if **SoftParityError** is connected back to the Event input). A hard error occurs if a location still causes a parity error on retry; in this case the processor is stopped immediately and the **HardParityError** pin is asserted.

After a hard parity error has been detected the debugger can be started in post–mortem mode. If the debugger fails to find a processor which has halted with the error flag set, it will try to find a T426 processor which has had a hard parity error.

It will then display this as the first processor in error. The debugger does not automatically locate to the program source if a parity error has occurred — the debugger will instead display the monitor page to allow the parity registers to be examined.

The parity registers are displayed on the monitor page at the bottom left of the display below the clock registers. These registers are *not* displayed in interactive mode. This is because the registers are volatile and reading the registers would interfere with any user code attempting to handle soft parity errors.

8.6.5 Debugging the root transputer

Programs which run on the root transputer, or which use the root transputer to run part of a multiprocessor program, must be debugged 'off-line' from a separately created memory image file. This is necessary because the debugger executes on the root transputer and overwrites the code in its memory. The memory dump is performed using the `idump` tool after the program has failed and before the debugger is started with the '`R`' option. Details of how to use the `idump` tool can be found in chapter 5 of the *Toolset Reference Manual*.

Skip loading

As an alternative to using the `idump` tool, the application program can be *skip* loaded onto the next processor on the network, avoiding the root transputer. This leaves the root transputer free to run the debugger. A disadvantage of this method is that it requires one extra processor on the network in addition to those needed for the program.

If only one transputer is available, for example on single-transputer boards, the memory dump method *must* be used. If more than one transputer is available skip loading is the recommended method since it enables the program to be directly debugged from transputer memory.

Use of the skip loader is described in chapter 7 of this manual and chapter 15 of the *Toolset Reference Manual*.

8.7 Interactive debugging

Interactive debugging allows programs to be executed under interactive control using breakpoints set in the code. Breakpoints can be set on any line of source. Symbolic and monitor page facilities can be used to examine code, inspect variables, jump down channels to other processes or processors, and determine the state of the network. Special symbolic functions and monitor page commands, only available in breakpoint mode, support the modification of variables and memory locations and the restarting of programs from the breakpoint or from other points in the code.

Programs that communicate to the host *must* use `iserver` protocol, as used by the standard I/O libraries, when being debugged interactively.

8.7.1 Runtime kernel

The breakpoint debugger places a special runtime kernel on each processor in addition to the application bootable code. This kernel provides a communication network to enable the debugger to transparently share transputer links with the application in addition to providing a breakpoint handler to deal with breakpoints, errors, inspection of processor state etc. The scheme is illustrated in Figure 8.2.

Note: The debugging kernel places the transputer into Halt-On-Error mode regardless of the error mode of the program. This means that during breakpoint debugging a transputer will always HALT when an error occurs.

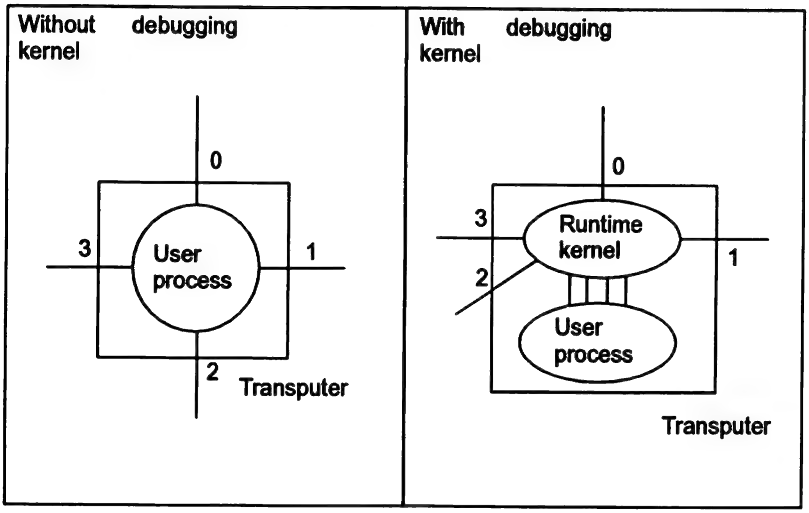


Figure 8.2 Debugger runtime kernel

The runtime kernel requires a certain amount of memory on each processor, the exact amount differing slightly between processor types. Kernels on processors with hardware support require slightly more memory because they retain more state information. The size of the kernel on each transputer type is given in table 8.2.

Apart from the extra memory required, the kernel is transparent to the application program if processes on different processors communicate with each other in the normal way, using channels supplied by the configurer.

Note: To allow breakpoint debugging to function correctly a program *must not* place channels explicitly onto processor link addresses. Programs that do so may introduce conflict with the runtime kernel, which also uses the links. Programs currently coded in this way should be re-coded to pass in hard channels, or edges, from the configurer, otherwise breakpoint debugging may not be used.

Processor	Kernel size	H/W support
M212	11.25K	No
T212	11.25K	No
T222	11.25K	No
T225	12.75K	Yes
T414	13.5K	No
T800	13.5K	No
T400	15.25K	Yes
T425	15.25K	Yes
T426	15.25K	Yes
T801	15.25K	Yes
T805	15.25K	Yes

Table 8.2 Runtime kernel size and processor breakpoint support

8.7.2 Processors without hardware breakpoint support

Certain transputers have built-in instructions to aid breakpointing (see table 8.2). For those processors without hardware breakpoint support, breakpoints should not be set within *high priority* processes because the mechanism used to implement breakpoints causes such processes to lock the processor and disables all communications to the processor via the runtime kernel. To help safeguard against this problem, the debugger monitor page breakpoint option will only set breakpoints at high priority process entry points or `main()` on processors with hardware breakpoint support.

The exact effect on the network of encountering such a breakpoint will depend on the position of the processor in the network hierarchy but the the possibility should be avoided. Since the debugger is, in general,unable to check the validity of breakpoints it is the programmer's responsibility to ensure correct operation on processors without direct hardware breakpoint support.

8.7.3 Creating programs for debugging

Programs to be debugged using breakpoint debugging should be compiled with full debug information enabled, using the C and FORTRAN compilers 'G' option and the occam compilers default.

All modules in the program must be compiled in the same, or a compatible, mode. Modes are checked at link time and if incompatible modes are found then the link is aborted.

The code must be produced *without* using the 'Y' option with any of the tools if interactive debugging is to be done.

8.7.4 Loading the program

Breakpoint debugging does not require special loading or memory dump procedures because the program is automatically skip loaded by the breakpoint debugger. However, breakpoint debugging does require one extra processor in the network because the root processor is dedicated to running the debugger.

Clearing error flags

If either `iserver` or `idebug` detect that the error flag is set immediately a program begins to run, it is likely that the network contains more processors than you are currently using, and that one or more of the unused processors has its error flag set. The error flag may be randomly set when the transputer is powered up — it is normally cleared by the bootstrap code.

The error flags of all the processors in a network can be cleared by running a network check program such as `ispy`. This ensures a clean network on which to load the program. This generally only needs to be done once, after the system is first turned on.

The `ispy` program is provided as part of the support software for some INMOS *i*q systems products. These products are available separately through your local INMOS distributor.

An alternative way of clearing all the error flags in the network is to load a dummy program which is configured to use every processor in the network. In the act of loading the dummy code the processor error flag is cleared.

Parity-checked memory

In system that include some processors which have external memory with parity-checking (e.g. systems built with the T426) it is necessary to initialize the contents of memory before the application code is run. This is because a read from un-initialized memory could cause a parity error to be reported.

Normally, when not breakpoint debugging, the contents of memory are initialized by the bootstrap loader code. This is controlled by the collector `CM` option (see chapter 3 of the *Toolset Reference Manual*).

The debugger has two command line options which can be used for for memory initialization — both of these are followed by a hexadecimal number representing the pattern to be written to memory. The '`J`' option writes the given pattern to all of the data areas (stack, workspace, static, heap and vectorspace as appropriate) in each processor. The '`K`' option writes to the same areas of memory as the '`J`' option and also to the 'freespace' area.

In general, the '`J`' option should be used for configured programs and the '`K`' option for non-configured programs (i.e. programs for a single processor produced using the collector's '`T`' option). The memory initialization is performed on all processors in the network, not just T426s.

These options can also be useful for seeing where data has been written to memory. For example, they can be used to determine the size of stack or heap used by a program when it runs, or to detect data written to unexpected areas of memory. Note that the bootstrap phase of a program may use a small part of the program data and freespace areas for its own purposes, consequently the pattern of data may have some holes in it.

8.7.5 Running the debugger

The command `idebug` starts the host file server program, `iserver`, to load the debugger onto the root transputer and provide it with host services. Different options need to be given to `idebug` depending on the type of debugging being done (e.g. breakpoint or post-mortem) and the details of the transputer network being used (e.g. is the code to be debugged running on the root processor or is that transputer available for the debugger. Some basic examples are given here.

Note that the transputer network is not reset or analyzed by default so, generally, one of the `iserver` options must be specified for this (e.g. '`SR`' or '`SA`'). This is true even if the '`D`' option is being used to run the debugger without using transputers (because the processor running the debugger must be reset).

When doing breakpoint debugging, the '`SR`' option is used to cause the `iserver` to reset the transputer network and the '`B`' option to specify which link from the root transputer is connected to the processors running the application code — for example:

```
idebug -sr -b 2 program.btl                    (UNIX)
idebug /sr /b 2 program.btl                    (MS-DOS and VMS)
```

As another example, when using the debugger in post-mortem mode to debug a program which does not use the root transputer, the '`SA`' option would be used to make the server put the network into Analyse mode with the '`T`' option to specify the link from the root processor to the transputer running the program to be debugged:

```
idebug -sa -t 2 program.btl                    (UNIX)
idebug /sa /t 2 program.btl                    (MS-DOS and VMS)
```

Finally, when debugging a program running on the root transputer in post-mortem mode, the `idump` command is first used to create a file containing a dump of the transputer's memory and then `idebug` is run with the '`R`' option to specify the core dump filename — for example:

```
idump core.dmp #100000

idebug -r core.dmp program.btl                 (UNIX)
idebug /r core.dmp program.btl                 (MS-DOS and VMS)
```

Complete details of which options to use in different circumstances are given in the *Toolset Reference Manual*, chapter 4.

8.7.6 Interactive mode functions and commands

Several symbolic debugging functions and monitor page commands are only available in interactive mode. The commands available are summarized below.

Symbolic functions

TOGGLE BREAK	Set or clear a breakpoint on the current line.
RESUME	Restart a process stopped at a breakpoint.
CONTINUE FROM	Restart a stopped process from the current line.
MODIFY	Change the value of a variable in memory.

Monitor page commands

B	Breakpoint menu.
J	Execute program.
S	Show debugging messages.
U	Update register display.
W	Write to memory.

8.7.7 Breakpoints

Breakpoints can be set, cleared, and listed using monitor page commands, and set/cleared using symbolic functions.

Breakpoints can be set at any point in a process running on any processor. At each breakpoint, or on program error, the process pauses and the source code may be displayed.

Note: When a process is stopped at a breakpoint or by one of the debugging functions (e.g. `debug_stop`) other parallel processes in the program continue to run. A side effect of pausing is that the debugger suspends `iserver` communications in order to preserve the debugger's screen display.

Breakpoints can be set at code entry points, or on any line of source code. Variables within scope at the breakpoint can be modified and the process restarted. Breakpoints can also be set at the monitor page but care should be taken not to set breakpoints at addresses that do not correspond to the start of a source code statement, otherwise the behavior is undefined.

Setting breakpoints at symbolic level is the recommended method.

8.8 Program termination

Program termination is signalled to the debugger by the termination of `iserver`. This is performed automatically by the C and FORTRAN runtime systems, and must be done explicitly by the user in occam code. If the program contains independently executing processes which do not require communication with the server the debugger may be resumed to interact with these processes.

To run or debug the program again it must be reloaded onto the transputer using **`iserver`**, or **`idebug`** in breakpoint mode.

8.9 Symbolic facilities

Symbolic debugging is debugging at source code level using the symbols defined in the program for variables, constants and channels. Features provided in symbolic debugging include the examination of source code, the inspection of variables and channels, and the backtracing of procedure calls. A number of special breakpoint functions are available if the debugger is run in breakpoint mode.

Source level debugging is accessed through symbolic *functions* mapped to specific keyboard function keys (e.g. $\boxed{\text{INSPECT}}$) by an 'ITERM' file. Keyboard layouts for specific terminal types can be found in the Delivery Manual that accompanies this release. Alternatively, the comments in the ITERM file can be read to find the mapping of functions to keys.

Help screen

A help page can be displayed by pressing either $\boxed{?}$ or $\boxed{\text{HELP}}$, this displays the following information:

```
                       idebug Symbolic Help Summary
                       ****************************

1-INSPECT 2-CHAN 3-TOP 4-RETRACE 5-RELOC 6-INFO 7-MOD 8-RESUME 9-MONITOR 0-BACK

The above list summarises the commonly used functions available in symbolic
mode.  For a complete list of all symbolic functions available please refer to
the idebug documentation.  The mapping of a symbolic function to a particular
key may be found in the file defined by the ITERM environment variable.

INSPECT    -  Display the type and value of a variable
CHANNEL    -  Locate the process waiting on a channel
TOP        -  Locate back to the error or last source code location
RETRACE    -  Undo a BACKTRACE
RELOCATE   -  Locate back to the last location line
INFO       -  Display process information (eg. Iptr, Wdesc, process name)
MODIFY     -  Change the value of a variable in memory
RESUME     -  Resume a process stopped at a breakpoint
MONITOR    -  Change back to the Monitor page
BACKTRACE  -  Locate to the calling function or procedure
HELP or ?  -  This help summary

========================  Hit any key to continue  ========================
```

The main symbolic debugging activities and the functions that are used to access them are described in the following sections.

8.9.1 Locating to source code

Locating to the source code for a particular process is a crucial procedure in the debugging process on which other operations depend. For each required location the debugger must be given a memory address which it uses to locate to the source. When the required code is located, symbolic functions can be used to browse the code and inspect variables. Where the source code is unavailable, for example, libraries supplied as object code with minimal debug information, the line containing the library call is located to instead.

When first started in post-mortem mode, the debugger determines the address of the last instruction executed, which it uses to automatically locate to the relevant source code. Subsequently for each new point to locate to in the code the debugger requires a new address which can be supplied by the programmer.

Process addresses can be determined using the monitor page $\boxed{\text{R}}$, $\boxed{\text{T}}$, and $\boxed{\text{L}}$ commands that display the processes waiting on the run queues, the timer queues, and the transputer links. To locate to a process displayed by one of these commands, use the $\boxed{\text{G}}$ command. Code corresponding to any memory address can be located using the monitor page $\boxed{\text{O}}$ command.

Certain addresses are already known to the debugger and can be located to using symbolic functions without specifying the address or switching to monitor page commands. Many of the common operations used during source code debugging can be performed directly with symbolic functions. They include relocating to the previous location, locating to the original error, and locating to a process waiting on a channel.

The symbolic functions that can be used directly for locating to specific locations and sections of source code are listed below.

$\boxed{\text{RELOCATE}}$	Locate back to the last location line.
$\boxed{\text{TOP}}$	Locate back to the error or last source code location.
$\boxed{\text{CHANNEL}}$	Locate to the process waiting on a channel.

The $\boxed{\text{CHANNEL}}$ function is described more fully in section 8.9.4.

Other functions which locate to specific sections of code are the $\boxed{\text{BACKTRACE}}$ and $\boxed{\text{RETRACE}}$ functions. These are used to trace subprogram calls and do not require a specified address. The functions are described in section 8.9.4.

A strategy for locating processes in multi-process programs is presented in section 8.11.

8.9.2 Browsing source code

Several functions are available for browsing source files once they have been located. They include functions for navigating files, changing to included or new files, and string searching. The functions are listed below.

TOP OF FILE	Go to the first line.
END OF FILE	Go to the last line.
GOTO LINE	Go to a specified line.
SEARCH	Search for a specified string.
ENTER FILE	Enter an included source file.
EXIT FILE	Exit an included source file back to the enclosing source file.
CHANGE FILE	Display a different source file.

8.9.3 Inspecting source code and variables

The values of constants, variables, parameters, arrays, and channels can be inspected at any point in the code. A special inspect function for channels allows the debugger to locate to the process waiting at the end of the channel. Symbols to be inspected must be in the scope of the source line last located to.

INSPECT	Display the type and value of a source code symbol.
CHANNEL	Locate to the process waiting on a channel.
TOGGLE HEX	Enables/disables Hex-oriented display of constants and variables. Selects the display of source code symbols in hexadecimal form for C and FORTRAN.
GET ADDRESS	Displays the start address of the sequence of transputer instructions corresponding to the selected source line.
INFO	Displays low-level information about the selected process.

8.9.4 Jumping down channels

The CHANNEL function can be used to locate to a process waiting on a channel. This is known as 'jumping down' a channel and works for channels on the same processor (internal or *soft* channels) or channels assigned in the configuration to transputer links (external or *hard* channels which connect processes on different processors together). It *cannot* be used to jump down software virtual links provided by the configurer. Debugging can then continue at the waiting process. If no process is waiting on a channel the channel is reported as '`Empty`'.

8.9.5 Tracing procedure calls

Two functions assist in the tracing of procedure and function calls. They can be used even if the source of the called routine is not present, for example, libraries supplied as object code with minimal debug information. In this case the line containing the function call is displayed rather than the library code itself. Where procedures are nested, successive backtrace operations will locate to the original call. Variables and other symbols can be inspected at any stage. The two functions are listed below.

BACKTRACE	Locate to the calling procedure or function.
RETRACE	Undo a BACKTRACE.

8.9.6 Modifying variables

The MODIFY function allows variables to be changed in transputer memory and the program continued with the new values. For C and FORTRAN it supports the same expression language as INSPECT. For further details see chapter 4 in the *Toolset Reference Manual.*

8.9.7 Breakpointing

Symbolic functions are provided for setting and clearing breakpoints, for modifying the value of a variable, and for continuing the program.

TOGGLE BREAK	Set or clear a breakpoint on the current line.
RESUME	Restart a process stopped at a breakpoint.
CONTINUE FROM	Restart a stopped process from the current line.
INTERRUPT	Force the debugger into the monitor page (without necessarily stopping the program).
MODIFY	Change the value of a variable in memory.

8.9.8 Miscellaneous functions

The following extra functions are available at symbolic level:

MONITOR	Change to the monitor page.
FINISH	Quit the debugger.

8.10 Monitor page

The debugger monitor page is a low level debugging environment which gives direct access to machine level data. It allows memory to be viewed and disassembled and gives access to information about the processor's activity through the display of error flag status and pointers to process queues. Specific debugging operations are selected by single letter commands typed after the '`Option`' prompt.

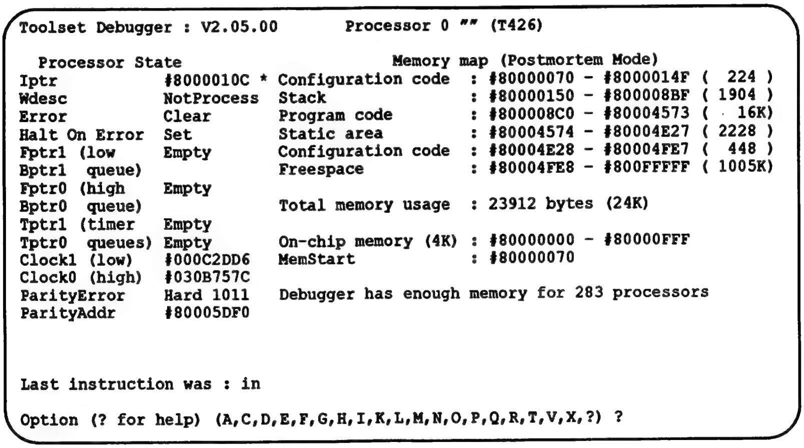
8.10.1 Startup display

When first started in interactive mode, or in post-mortem mode with an invalid **Iptr** or **Wdesc** (see below), the debugger enters the monitor page environment and displays information such as the addresses of instruction and workspace pointers, status of error flags, and information about the processor run queues. The memory map is also displayed.

If an **Iptr** or **Wdesc** is invalid at startup it is indicated by an asterisk ('*'). A double asterisk ('**') is used to indicate an **Iptr** or **Wdesc** which is outside the defined memory on a processor (i.e. beyond the 'freespace').

The monitor page display differs slightly between post-mortem and breakpoint modes. In post-mortem mode the display includes the saved pointers for the low priority process if the processor was running at high priority when analyzed; in breakpoint mode the display does not include these pointers but does include the contents of the registers **Areg**, **Breg**, and **Creg**, if known. At startup in breakpoint mode, no machine pointers or register values are available (the program has not yet started) and so no values are displayed.

A typical startup display is shown in Figure 8.3.



```
Toolset Debugger : V2.05.00      Processor 0 "" (T426)

  Processor State                        Memory map (Postmortem Mode)
Iptr            #8000010C * Configuration code   : #80000070 - #8000014F (  224 )
Wdesc           NotProcess  Stack                : #80000150 - #800008BF ( 1904 )
Error           Clear       Program code         : #800008C0 - #80004573 (   16K)
Halt On Error   Set         Static area          : #80004574 - #80004E27 ( 2228 )
Fptr1 (low      Empty       Configuration code   : #80004E28 - #80004FE7 (  448 )
Bptr1  queue)               Freespace            : #80004FE8 - #800FFFFF ( 1005K)
Fptr0 (high     Empty
Bptr0  queue)               Total memory usage   : 23912 bytes (24K)
Tptr1 (timer    Empty
Tptr0  queues)  Empty       On-chip memory (4K)  : #80000000 - #80000FFF
Clock1 (low)    #000C2DD6   MemStart             : #80000070
Clock0 (high)   #030B757C
ParityError     Hard 1011   Debugger has enough memory for 283 processors
ParityAddr      #80005DF0



Last instruction was : in

Option (? for help) (A,C,D,E,F,G,H,I,K,L,M,N,O,P,Q,R,T,V,X,?) ?
```

Figure 8.3 Example post-mortem startup display for a T426 processor

Items displayed on the startup page and their meanings are summarized in table 8.3. Most of the data displayed is common to all transputer types. Where the display differs for specific processor types and debugging modes, this is indicated in the table.

Item displayed	Description
`Iptr`	Instruction pointer (address of the last instruction executed).
`Wdesc`	Process descriptor (process priority and workspace pointer).
`IptrIntSave`†	Saved low priority instruction pointer, if applicable.
`WdescIntSave`†	Saved low priority process descriptor, if applicable.
`A Register`‡	Contents of A register, if known.
`B Register`‡	Contents of B register, if known.
`C Register`‡	Contents of C register, if known.
`Error`	Status of transputer error flag.
`FPU Error`	Status of FPU error flag (T80x series only).
`Halt On Error`	Status of halt on error flag.
`Fptr1`	Front pointer to low priority process queue.
`Bptr1`	Back pointer to low priority process queue.
`Fptr0`	Front pointer to high priority process queue.
`Bptr0`	Back pointer to high priority process queue.
`Tptr1`	Pointer to low priority timer queue.
`Tptr0`	Pointer to high priority timer queue.
`Clock1`	Value of low priority transputer clock.
`Clock0`	Value of high priority transputer clock.
`ParityError`†	Status of parity error register, if applicable.
`ParityAddr`†	Address of parity error, if applicable.
† Not available in breakpoint mode. ‡ Not available in post–mortem mode. Not known to the debugger in break-point mode on processors with no hardware support for breakpointing.	

Table 8.3 Data displayed at the monitor page

Process Workspace or Stack

A process workspace (or stack) consists of a vector of words in memory. It is used to hold local variables of the process. The workspace is organized as a falling stack, with 'end of stack' addressing; that is the local variables of a process are addressed as positive offset from the workspace pointer (**Wptr**).

Process Descriptors

In order to identify a process completely, it is necessary to know both its workspace pointer **Wptr** (in which the byte selector is always 0), and its priority (which is 0 for high priority and 1 for low priority). A process descriptor, **Wdesc**, is the sum of the process's workspace pointer, **Wptr**, and its priority.

Process pointers

Iptr points to the last instruction executed and **Wdesc** contains the process descriptor. The saved low priority **Iptr** and **Wdesc** are also displayed if the processor was running a high priority process when it was halted. An asterisk placed next to either an **Iptr** or **Wdesc** indicates an invalid memory location for the process. A double asterisk indicates that the address is outside the defined memory map of the processor. A **Wdesc** value of '**NotProcess**' indicates that no process was executing on the processor when it halted

Practical notes:

- If **Wdesc** contains the value '**MemStart**' it is likely that the **Analyse** signal has been asserted more than once on the network. This can occur on transputer boards where the subsystem signal is asserted on analyze, as on the IMS B004. For further guidance on the use of such boards refer to chapter 4 in the *Toolset Reference Manual*.
- If **Wdesc** contains the word '**NotProcess**' it means that there were no runnable processes at that instant on the transputer (check timer and external links for any waiting processes) — this may also occur in the presence of deadlock.
- If **WdescIntSave** contains the word '**NotProcess**' it means that a low priority process was not interrupted when the high priority process started running.

Fptr and **Bptr** point to the process run queues, which hold information about processes awaiting execution. The suffix **0** indicates the high priority queue and the suffix **1** indicates the low priority queue.

If the front and back pointers are the same then only one process is waiting; if there are no processes waiting the pointers have no value and the queue is shown as 'Empty'.

Tptr0 and **Tptr1** are pointers to the high and low priority timer queues respectively.

Registers

In breakpoint mode only, the contents of the transputer registers **Areg**, **Breg**, and **Creg** are displayed for those processors which have built in instructions for breakpoint handling (see table 8.2). Values displayed are those which were current when the process stopped.

Error flags

Two flags are displayed for all processors: **Error** and **HaltOnError**. The **FPError** flag is also displayed for transputers with an integral floating point unit (IMS T80x series).

Clocks

Clock0 and **Clock1** display the values of the high and low priority clocks when the process was stopped. In breakpoint mode the clock values, queue pointers and link information can be updated using the monitor page $\boxed{U}$ command.

Parity errors

ParityError and **ParityAddr** are only displayed for a T426 processor in post-mortem mode. **ParityError** is the state of the **ParityErrorReg** and can contain one of the following:

Soft *xxxx*	A soft parity error has occurred
Hard *xxxx*	A hard parity error has occurred

The value *xxxx* shows the byte selector bits of the error registers; the value is in binary with byte 3 on the left through to byte 0 on the right. Thus, the value 1011 would show that bytes 3, 1, and 0 are in error.

NotInMem	The memory in a dump file does not include the parity registers
Clear	No parity error has occurred

ParityAddr shows the state of the **ParityErrorAddressReg** and can contain one of the following:

#hhhhhhhh	Word address, in hexadecimal, of location where error occurred
NotInMem	The memory in a dump file does not include the parity registers
Undefined	No parity error has occurred

Memory map

The memory map display is included on the standard startup display — this is the same memory map as displayed by the monitor page $\boxed{M}$ command. Any or all of the following memory segments may be displayed, depending on the application program and its configuration:

Runtime kernel
Reserved memory
Configuration code
Stack (Workspace)
Program code
Vectorspace
Static area
Heap area
Configuration code
Freespace

When the memory map is displayed, the mode that the debugger is running in is shown. This will be one of:

`Interactive Mode`	When interactively debugging a program.
`Postmortem Mode`	When debugging a program in post-mortem mode.
`Interactive Postmortem`	When post-mortem debugging a program which was previously debugged interactively.
`Dummy Session`	When the debugger is started with the `D` command line option.

8.10.2 Monitor page commands

Most monitor page commands are single-letters that are typed at the monitor page `Option` prompt. A few commands are mapped onto specific function keys. The commands that support breakpoint debugging are only available when the debugger is run in interactive mode.

The main monitor page commands allow you to disassemble and display transputer memory, locate and debug processes, and examine the network processor by processor.

The main commands for common debugging operations are introduced in the following sections. Full details of all the commands can be found in chapter 4 of the *Toolset Reference Manual.*

Examining memory

Specific segments of transputer memory can be displayed in hexadecimal, ASCII, any high level language type, or disassembled into transputer instructions. The segment of memory to be displayed is specified by a starting address. A map of the transputer's memory can be displayed giving the positions of code and workspace. Commands for examining transputer memory are summarized below.

A	Display memory in ASCII.
D	Disassemble into transputer instructions.
H	Display memory in hexadecimal.
I	Display memory in selected data type.
M	Memory map.

Locating processes

Locating to code for specific processes is one of the major functions available through the monitor page. The commands available allow processes other than the stopped or current process to be located and examined anywhere on the network. Processes can be located on the current processor by examining run queues, and on other processors by jumping down transputer links.

Four commands are used, three to display waiting processes and one to jump to the selected code of a process displayed by the other three.

$\boxed{R}$	Display processes waiting on Run queues.
$\boxed{T}$	Display processes waiting on Timer queues.
$\boxed{L}$	Display processes waiting on transputer Links.
$\boxed{Z}$	Display processes waiting on software virtual links.
$\boxed{G}$	Goto symbolic debugging for the selected process.

These commands can be used to trace all processes on a network and determine the cause of program failure. The method is explained in more detail in section 8.11.

Specifying processes

The $\boxed{O}$ command allows a specific process to be selected for symbolic debugging, providing the address is known.

$\boxed{O}$	Specify a process for symbolic debugging.

This command is useful for switching directly to symbolic debugging for a process whose instruction pointer and process descriptor you have already noted, earlier in the debug session.

Selecting processes

The $\boxed{F}$ command enables a source file to be selected for symbolic display using the filename of the *object* module produced for it.

$\boxed{F}$	Select a source file to be displayed.

This option enables symbolic locating (for setting breakpoints etc.) without needing to know `Iptr` and `Wdesc` process details (as the $\boxed{G}$ and $\boxed{O}$ commands do).

Other processors

Two commands and two cursor keys allow other processors to be selected.

$\boxed{E}$	Go to next halted processor.
$\boxed{P}$	Go to specified processor.
$\boxed{\blacktriangleleft}$	Go to the next lowest numbered processor.

▶	Go to the next highest numbered processor.

The sequence of processors used by the E and cursor key commands is an internal sequence read by the debugger. Processor numbers corresponding to visible names in the configuration file can be determined by using the K command.

Breakpoint commands

The following commands support breakpointing. To use these commands the debugger must be run with the '`B`' command line option.

B	Breakpoint menu.
J or RESUME	Jump into and run application program.
S	Show debugging messages and prompts menu.
U	Update processor status display.
W	Write value to memory.

Changing to post–mortem debugging

When a program crashes during interactive debugging you are able to change to post-mortem debugging using the following command:

Y	Postmortem debug current breakpoint session.

8.11 Locating processes

Most transputer programs consist of several processes running in parallel, either on the same transputer or on separate processors connected by their INMOS links.

If a program error halts the transputer then the debugger automatically locates to the stopped process, which can then be examined directly. If the program runs incorrectly but does not halt the processor, a good approach is to locate to and examine each process in turn.

There may be many processes running on the transputer when it is interrupted from the keyboard, or the `idump` tool is run to create a dump file for debugging. Each process exists in one of a number of possible states:

- Not yet started.
- Running on the processor.
- Waiting on a process execution queue (Run queue).
- Waiting on a timer queue.
- Waiting for communication from another process on the same processor.
- Waiting for communication on a transputer link (Link information).
- Interrupted by a high priority process.
- Already stopped or terminated.

8.11.1 Running on the processor

One, and only one, process may execute on the transputer at any instant. The debugger will automatically locate to this process (if there was one) when the debugger is executed. All other processes are either waiting, stopped, or not yet started.

8.11.2 Waiting on a run queue

Processes on the run queues (i.e. waiting to be executed) can be located by first using the monitor page R command to display the list of waiting processes. A process can be selected from the list by pressing G (for 'Goto process'), moving the cursor to the appropriate address, and then pressing RETURN. Processes can also be located to by specifying the displayed `Iptr` and `Wdesc` with the O command.

The values displayed with the R command can be used to determine the overall status of run queues. If no processes are waiting then the content of the queue is shown as '`Empty`'. If pointer addresses are displayed then there are processes waiting; if the front and back pointers have the same value then there is only one process waiting.

8.11.3 Waiting on a timer queue

Processes waiting for a specified time are placed on the high and low priority timer queues. These are similar to the run queues except that they are controlled by the transputer clocks.

In a similar way to processes on the Run queues, processes on the timer queues can be located by using the monitor page T command to display a list of processes and then using the G command, or by specifying the process address. Pointers to the timer queues indicate overall queue status in a similar way to the run queues.

8.11.4 Waiting for communication on a link

Processes waiting for a hardware communication (input or output on a transputer link, or an input on the Event pin) can be located by using the monitor page L command to display a list of waiting processes, and then using the G command to locate to the process. Links where no processes are waiting are shown as '`Empty`'.

At most 9 processes can be waiting for a hardware communication, two for each of the four links and one on the Event pin.

See section 8.4.1 for information on the restrictions on locating down hard channels.

8.11.5 Waiting for communication on a software virtual link

Processes waiting for a communication on a software virtual link (as provided by the configurer) can be located by using the monitor page $\boxed{\text{Z}}$ command to display a list of waiting processes, and then using the $\boxed{\text{G}}$ command to locate to the process. Virtual links where no processes are waiting are shown as '**`Empty`**'.

This is the preferred method for locating processes waiting on external communications when software virtual links are present.

8.11.6 Waiting for communication on a channel

Processes waiting for a communication on a channel can be located from source level using the $\boxed{\text{CHANNEL}}$ function. This function works for both internal (or *soft*) channels and external (or *hard*) channels (channels mapped onto processor links).

Only one process can be waiting on a channel. If no process is waiting, the channel is shown as '**`Empty`**'.

8.11.7 Interrupted by a high priority process

A low priority process may have been interrupted by a high priority process. Such a process may be selected using the $\boxed{\text{G}}$ or $\boxed{\text{O}}$ commands and the values stored in the **`WdescIntSave`** location.

8.11.8 Processes terminated or not started

Processes which have stopped executing, or not yet started, do not have process descriptors and so they cannot be examined by the debugger. If the currently running process and all the waiting processes have been found (not forgetting all those processes waiting on all the internal channels) then any processes still unaccounted for must either have already finished or failed to start.

8.11.9 Locating to procedures and functions

When a procedure is called, the workspace pointer is moved. If the debugger locates inside any code of defined scope (such as a procedure) then only local variables, and variables declared globally, are in scope and available for inspection.

To inspect variables or channels not in scope within the procedure or function, use $\boxed{\text{BACKTRACE}}$ to locate to a position where the desired variable or channel is in scope. To relocate back into the procedure or function use $\boxed{\text{RETRACE}}$ to undo each backtrace, or $\boxed{\text{TOP}}$ to return to the initial location.

8.12 Debugging support library

Three routines are provided in the libraries to assist with debugging. These provide the functions *stop*, *assert*, and *message*. The routines have different names for each language and are described in more detail in the appropriate *Language and libraries* manuals. Table 8.4 summarizes the routines for each language. The descriptions and examples below use the C versions of these functions.

Routine		Description
`debug_assert`	C	If the parameter evaluates to false then stop the process and inform the debugger.
`DEBUG.ASSERT`	occam	
`DEBUG_ASSERT`	FORTRAN	
`debug_stop`	C	Stop the process and inform the debugger.
`DEBUG.STOP`	occam	
`DEBUG_STOP`	FORTRAN	
`debug_message`	C	Insert a debugging message in the program.
`DEBUG.MESSAGE`	occam	
`DEBUG_MESSAGE`	FORTRAN	

Table 8.4 Debug support functions

The stop and assert routines are used to stop a process, the latter on the failure to meet a specified condition; such events are treated as a program error by the debugger. The message is used to insert messages that will only be displayed when the program is run under the interactive debugger.

For C and FORTRAN the procedures are included in the standard library that is incorporated at link time and are directly accessible from the program without further action by the programmer. For occam programs, the library `debug.lib` must be referenced with a `#USE` in the source code and also included as an input to the linker.

`debug_assert()` and `debug_stop()` allow a process to be stopped at any point in the code, where it can then be debugged using the symbolic functions and Monitor page commands. `debug_stop()` always stops the process whereas `debug_assert()` only stops the process if the parameter evaluates to false.

`debug_message()` is used to insert debugging messages into the code. Messages are relayed back to the terminal from any point in the program, even from code running on distant processors of a network. It can be used to monitor the activity of outlying processors which are not directly connected to the host. The display of debug messages at the terminal is controlled by an option on the Monitor page Breakpoint Menu (the default is to display them).

Note: Only the first 80 characters of a message will be displayed.

Example

The use of the debug support functions is illustrated in the example below. There is an occam version with a similar structure. Both examples may be found in the debugger examples directory.

```
/*****************************************
*
*  Debugger example:  debug.c
*
*  Example of debug support functions when used with
*  and without the debugger.
*   (see also debugger example file abort.c)
*
*****************************************/

#include <stdio.h>
#include <stdlib.h>
#include <misc.h>

int
main (void)

{
       /* 0 will cause assertion to fail */
        int     x = 0;

        printf ("Program started\n");

        debug_message("A message only within the debugger");

        printf ("Program being halted by debug_assert()\n");
        debug_assert (x);

        printf ("Program being halted by debug_stop()\n");
        debug_stop ();

        exit (EXIT_SUCCESS);
}
```

In this example, if **x** is 1 `debug_assert` evaluates to true and the program runs until it encounters `debug_stop`. If **x** is 0 (as in the example) `debug_assert` evaluates to false and the process stops before it reaches `debug_stop`. Code stopped by `debug_assert` and `debug_stop` may be resumed from the line following the call of the debug function using the CONTINUE FROM key.

8.12.1 Action when the debugger is not available

If the debugger is not available on the system the debug library procedures have the following actions:

`debug_assert`	If the parameter evaluates to false then stop the process (also stops the processor if configured in HALT mode).
`debug_stop`	Stop the process (also stops the processor if configured in HALT mode).
`debug_message`	No action.

8.13 Debugging with `isim`

The T425 simulator `isim` provides a single processor interactive simulation of a program running on an IMS T425 transputer, on a boot from link transputer board connected to a host computer through the host file server `iserver`. The interactive environment provides a machine level (non-symbolic) environment, similar to the debugger monitor page, for debugging programs and monitoring program execution.

The simulator allows any single processor program to be run and analyzed without a transputer board. All the component parts of a program to be simulated, must be compiled for the T425 transputer type (or compatible class — see appendix B of the *Toolset Reference Manual.*).

Note: The simulator can only be used to simulate single transputer programs.

8.13.1 Command interface

The simulator has a single command interface which corresponds to the debugger monitor page. Most commands are single letter commands and can be executed with a single key press. For a list of commands see chapter 14 in the *Toolset Reference Manual.*

8.13.2 Using the simulator

The simulator can be used in two ways:

- To debug programs by inspection of the transputer and memory, in the same way as with the debugger. Registers, memory, and machine state can be examined directly at the monitor page.
- To monitor the execution of programs using machine level single step execution and the setting of break points at specific memory locations. Code can be executed by stepping single transputer instructions.

8.13.3 Program execution monitoring

The simulator provides a number of functions that can be used interactively to monitor and control the behavior of a program. These are:

- Breakpoints
- Single step execution of a program

Breakpoints

Breakpoints can be set, displayed, and cancelled using the '`B`' command to display the Breakpoint Options Page.

Single step execution

A program can be stepped a single transputer instruction at a time using the 'S' command.

8.13.4 Core dump file

`isim` may be used to produce a core dump file that can be read by the debugger (as if the code had been executed on a real transputer and the memory dumped using the `idump` tool).

8.14 Hints and further guidance

This section gives some further guidance on some specific points related to use of the debugger.

8.14.1 Invalid pointers

The debugger checks process instruction pointers (`Iptr`) and process descriptors (`Wdesc`) for the correct code and data limits. Invalid pointers are flagged by an asterisk (`*`) on the screen. Invalid pointers outside the processor's memory are flagged with a double asterisk ('`**`').

Invalid pointers can indicate a major problem with the program. They may also be caused by specifying an incorrect dump file.

8.14.2 Examining and disassembling memory

Within the monitor page environment, the debugger keeps a record of two memory addresses; the start address of the last disassembly, used as the default by the $\boxed{D}$ command, and the address of the last region of memory to be displayed, used by the $\boxed{A}$, $\boxed{H}$ and $\boxed{I}$ commands.

This allows you to switch easily between code disassembly and memory display. You can, for example, disassemble a portion of memory using the $\boxed{D}$ command, examine its workspace in hex using the $\boxed{H}$ command, and then return to the original address by using the $\boxed{D}$ command once again.

8.14.3 Scope rules

The debugger can only display variables that are in scope at its current location point in the source code.

8.14.4 Inspecting soft configuration channels

Soft channels declared at the configuration level (i.e. those internal to a processor which are not placed on its external links) may be inspected from the monitor page by knowing that they are located near the beginning of the *Configuration code* area which appears after the user *Program code* area (as displayed by the monitor page Memory map command).

8.14.5 Locating to IF, ALT and CASE in occam

`IF` and `ALT` constructs with no `TRUE` guards, and `CASE` constructs where no selections are matched, stop the program as though a `STOP` statement had been encountered. In cases like these there is no obvious statement to locate to and the debugger locates instead to the *start* of the construct.

When using these constructs it is good practice to always define the default case. The debugger can then locate directly to the `STOP` statement where the error occurred.

8.14.6 Analyzing deadlock

Deadlocks that occur in multitransputer networks can be debugged by using the Monitor page '`L`' command to examine processes on the transputer links. Deadlocks in single transputer programs are more difficult to debug because there is no way to enter the program; there are no active processes from which to inspect channels, and no links to other transputers to provide an alternative entry point.

In practice, it is often obvious to the programmer which channel or channels are causing deadlock, and a dummy process can be added to the program to provide an entry point for the debugger. This is illustrated below using occam code for brevity; similar programs could be written in C or FORTRAN.

Consider the following code which creates a deadlock:

```
------------------------------
--
--  Debugger example:  deadlock.occ
--
--  Example of deadlock.
--
------------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"


PROC deadlock.entry (CHAN OF SP fs, ts, []INT
free.memory)

  PROC deadlock ()
    CHAN OF INT c :
    PAR
      SEQ
        c ! 99
        c ! 101

      INT x :
      SEQ
        c ? x
  :                --  <==  Missing second input

  SEQ
    deadlock ()
    so.exit (fs, ts, sps.success)
:
```

The program can be debugged by adding a process that will remain idle (here, waiting on a `TIMER`) while the program is debugged. An example of the type of code that is required is illustrated below.

```
------------------------------
--
--  Debugger example:  deadfix.occ
--
--  Example of deadlock and how to provide
--  debugging support.
--
------------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"
#USE     "debug.lib"


PROC deadfix.entry (CHAN OF SP fs, ts, []INT free.memory)

  PROC deadlock.debug ()
    CHAN OF INT c :
    CHAN OF INT stopper :
    PAR
      DEBUG.TIMER (stopper)    --  Hook for debugger
      SEQ
        PAR
          SEQ
            c ! 99
            c ! 101

          INT x :
          SEQ
            c ? x
            --  <==  Missing second input

        stopper ! 0  --  terminate debug.timer
  :

  SEQ
    deadlock.debug ()
    so.exit (fs, ts, sps.success)
:
```

The procedure `DEBUG.TIMER` is supplied in the occam debugging library. Similar routines could be written for other languages, and the principle of operation is the same – the process lies dormant on the processor's timer queue waiting for a time as far into the future as the processor can provide. When the timeout expires, the process places itself back on the timer queue. Such a process provides a hook into the program for locating deadlocked processes because the process is always accessible to the debugger on the timer queue. By locating to it you can access variables which are in scope at the point of its execution and thereby detect the deadlock. In the modified program a deadlock still forms in the procedure, but there is now a way to enter the program.

To enter the program and inspect the deadlock, first invoke the Monitor page environment, and use the Monitor page 'T' command to inspect the transputer's timer queue, on which there will be a process waiting. Use the 'G' command to go to that waiting process, and the debugger will locate to the call of `DEBUG.TIMER`.

You can then use [INSPECT] to examine the channel `c` where the program has deadlocked, and which will therefore contain the process that is waiting for communication. Finally you can use [CHANNEL] to jump to the deadlocked process.

The compiler does not insert this kind of debugging code automatically, for several reasons. Firstly, it is the philosophy of the toolset that the runtime code should not be needlessly altered. Secondly, most programs use many channels, and the execution overheads and code size could become unacceptably large. Again for the above example code this would be unimportant because the process consumes no CPU time, but this may not always be true. Lastly, it could be difficult to distinguish the true deadlocked process from the many idle debug processes waiting on the timer queues.

8.15 Points to note when using the debugger

This section contains some extra information which may be of use when using the debugger.

8.15.1 Abusing hard links

Current generation transputers permit unsynchronized transfer of messages on external channels (links). This allows, for example, two 4-byte messages to be sent and for them to be received as a single 8-byte message on the receiving transputer. This is not consistent with the communication of messages between processes on the same processor where the transfer of messages is synchronized.

When breakpoint debugging, external communications are handled by the debugger's virtual link system; this involves an internal transfer which will function incorrectly if user code is relying on unsynchronized transfers. Unsynchronized data transfer should not be used where breakpointing is used to debug a program. It is bad practice anyway and will certainly cause the virtual link system (used by both the debugger and the virtual-routing configurer) to crash.

8.15.2 Examining an active network (the network is volatile)

When a process stops at a breakpoint you should remember that all of the other processes are still running (unless they hit a breakpoint, terminate etc.). This means that data displayed by any of the monitor page commands that display process queues, etc. (e.g. R, L, T etc.) may change if they are re-displayed (e.g. by using the same command again or the U, Update, command to update the displayed information).

When in symbolic mode the same is true for channels which may appear empty when first inspected only to change to a waiting process when inspected again. The only way to effectively *freeze* all processes is to flip to post-mortem mode by using the monitor page Y (Enter Postmortem Mode) command. You should remember that when you use this command that all processes that have hit a breakpoint will not appear in the runtime queues. If this is a problem, you should note the `Iptr` and `Wdesc` values of the processes and, when in post-mortem mode, use the monitor page O (Select Process) command to locate to them symbolically.

8.15.3 Using INSPECT with channel communications

When debugging a program compiled for interactive debugging it should be remembered that channel communication is achieved via library calls. As a consequence, the INSPECT key may display an `Iptr` relating to code in the debugging kernel system rather than the `Iptr` of a user process waiting on the channel. This may lead to several channel communications appearing to having the same process `Iptr` (the `Wdesc` will be valid and unique). In order to correctly establish the `Iptr` of the process waiting at the other end, you should use the CHANNEL key to locate to the process followed by the INFO key to obtain process details.

8.15.4 Debugging in the presence of software virtual links

When the configurer creates software virtual links it places additional processes onto the processor in order to provide the virtual link services. These processes will be displayed by the debugger — it displays all processes it finds on the run queue, links etc. A consequence of this is that, occasionally, a process will be displayed which forms part of the software virtual link system. It is not possible locate to these processes (as they are is not part of the program being debugged). These processes may be identified by noting the `Iptr` and `Wdesc` values and using the V command to search for a process with a code area which contains the `Iptr` value, and a stack area which contains the `Wdesc` value. If the name of the process is "`%ROUTER[]`" then it is a software virtual link process which you may not locate to.

A similar problem occurs when attempting to locate to a process waiting on a transputer link which is used by the software virtual link system — the debugger will complain that it cannot find a file with a name such as "`vrdeb`*xx*`.tco`" (where *xx* is a sequence of digits).

Another problem encountered with using software virtual links and `idebug` is that low priority user processes are promoted, temporarily, to high priority when they communicate on software virtual links The debugger cannot tell if they were originally at high or at low priority: it will locate to what it believes is a high priority process. In general, this is not a problem if you wish to inspect variables etc. If this does present a problem and you know that a particular process is a low priority process, you should use the O command and specify a low priority `Wdesc` when prompted, by setting the least significant bit of the `Wdesc` value of the process (e.g. `%1234` becomes `%1235`).

In general, the preferred method for locating processes waiting on external communications when software virtual links are present is the Monitor page Z command. If however, you know that a transputer link is not used for software virtual routing, you should use the Monitor page L command to locate to such processes.

8.15.5 Selecting events from specific processors

The debugger provides no guarantee that debugging events, such as breakpoints and debugging messages, from processes running on different processors are presented in the same that order they occur in. Events on processors which are closer, in terms of connectivity, to the root transputer (where the debugger is running) are usually displayed before events on more distant processors.

If it is important that you encounter a debugging event on a specific processor before events on other processors, you can usually achieve this by changing to the processor of interest (using the monitor page P command or left and right cursor keys) *before* resuming via the J or RESUME command.

8.15.6 Minimal confidence check

A first level confidence check to perform with a program which is misbehaving is to perform a 'compare memory' check using the monitor page C command. This will help to highlight any memory corruption problems which may occur due to faulty memory or faulty program logic. If using occam, you can prevent out of range accesses to memory by ensuring that no compiler checks have been disabled.

8.15.7 INTERRUPT key

The debugger can be diverted from the running program to return to the monitor page by the use of the INTERRUPT key. However, problems can arise if the running program is simultaneously trying to read from the keyboard; the debugger is then unable to intercept the interrupt key. (Sometimes it is possible to force the interrupt to be recognized by repeating the key quickly.)

A similar problem arises when there are existing keystrokes buffered before the interrupt key; if the application program does not read these buffered keystrokes the debugger will never have a chance to see the interrupt key.

Note: The INTERRUPT key will disable all `iserver` requests to the application until the debugger is directed to resume the application.

8.15.8 Program crashes

If the debugger detects that the program has crashed immediately after starting program execution (i.e. after the J, Jump into application, command), you should use the post-mortem debug command, Y, to determine the cause. However, if no error flags are set on the network that is running the program then it is likely that an error flag is set on a transputer that is not in use. This may occur on boards where the subsystem services are wired to propagate all error flags to the root transputer. In this instance you need to clear all the error flags in the network (see section 8.7.4).

8.15.9 Undetected program crashes

When operating in breakpoint mode and a program overwrites the debugging kernel or you have set a breakpoint in a high priority process on a processor without hardware breakpoint support, the debugger cannot fully recover and is unable to indicate that the program has crashed. In this situation the debugger fails to update the screen other than to put the following message at the top of the screen when it attempts to display the monitor Page:

```
    Toolset Debugger : V2.05.00 Processor n "name" (Txxx)
```

In such instances you should use the host BREAK key in order to terminate the debugger and restart the debugger using the command line '`M`' option to post-mortem debug the session.

8.15.10 Debugger hangs when starting program

If the debugger hangs immediately after you have supplied the command line arguments when starting execution of a program you have probably set a breakpoint in a configuration-level, high priority process on a processor without hardware breakpoint support.

8.15.11 Debugger hangs

If the debugger hangs when attempting to flip to post-mortem mode using the monitor page Y command, or when trying to quit, you should terminate the debugger manually using the host BREAK key. If you were trying to switch to post-mortem mode you should restart the debugger using the command line '`M`' option to resume debugging in post-mortem mode.

8.15.12 Catching concurrent processes with breakpoints

Sometimes a concurrent process is executing in a program (often in a loop) and you would like to be able to control it better by using breakpoints. If the process is communicating with other processes via channels, and you have set breakpoints in these other processes, then breakpoints can be set on a communication and, when you hit that breakpoint, the channel can be jumped down to debug the executing process.

However, if the process has entered a non-communicating loop or you are not sure where exactly it is in your program code, you must use a different approach. In order to set a breakpoint, you should use the INTERRUPT key to return to the monitor page and then, by using the R (Run queues) command and/or the T (Timer queues) command, list the `Iptrs` and `Wdescs` of the processes currently executing. (Often, this will include the debugging kernel processes but these are easy to detect because they are marked as kernel processes.)

Use the G (Goto process) command to select the `Iptr` and `Wdesc` to locate symbolically to the process. You can then set a breakpoint on that line, return to the monitor page and resume the program using the J or RESUME command; when the process hits the breakpoint you may continue to debug it. If there are no processes on either the run or timer queues and there are no external communications, it means that your program has either deadlocked or terminated.

8.15.13 Phantom breakpoints

Because of the mechanism used for breakpoints on those transputers without hardware breakpoint support (see table 8.2) it is possible for the output from the INMOS compilers to contain code that fools the debugger into thinking it is a breakpoint (a *phantom* breakpoint). This happens when the code contains an empty loop that does not terminate. The following code examples will generate phantom breakpoints:

C	FORTRAN	occam

`while (1){`
`      ;`
`}`
	`DO WHILE TRUE`
	`END DO`
		`WHILE TRUE`
		`  SKIP`

`for (;;){`
`      ;`
`}`
	`100   GOTO 100`	

If you encounter a phantom breakpoint and you wish to continue execution, you must set a breakpoint at the same address and then resume execution. To do this use the GET ADDRESS key to obtain the start address of the empty loop when in symbolic mode, change to the monitor page and use the Set Breakpoint option on the Breakpoint menu to set a breakpoint at the loop address.

8.15.14 Breakpoint configuration considerations

When breakpoint debugging you should remember that the root transputer of a network is used by the debugger for its own purposes. On some transputer motherboards with an built-in pipeline, the root transputer is normally booted down link 0; subsequent transputers in the pipeline boot down link 1. This may (accidentally) be a problem if you simply take a network configuration which was not configured with breakpoint debugging in mind (e.g. a pipeline configuration) and attempt to breakpoint debug it. The debugger will in effect, attempt to skip load it onto the rest of the network; the program may load (if by chance the right link connections are available) but, if the boot link is different, it will not be able to talk to the host (via `iserver`) when it executes.

Such a problem may easily be checked for by using the monitor page $\boxed{\text{L}}$ command when positioned on processor 0. This will indicate whether the root transputer was booted from a different link to that specified in the configuration file.

When breakpoint debugging, the debugger will warn you if the boot link is different from that expected for the root processor before the network is loaded.

8.15.15 Determining connectivity and memory sizes

In order to establish the connectivity and memory map range for each processor in a program you should use the `icollect` '`P`' option. Alternatively you may use the debugger command line option '`D`' (dummy debug).

8.15.16 Long source code lines

Source code lines longer than 500 characters cause the symbolic source code browser to treat them as multiple lines and subsequently it will loose line synchronization; (i.e. it displays incorrect line number information).

8.15.17 Resuming breakpoints on the transputer *seterr* instruction

If an attempt is made to resume from a breakpoint which is at the address of a *seterr* instruction, the debugger does not continue with the original (correct) `Iptr` (it resumes with an `Iptr` within the kernel area). Because the debugger operates in Halt-on-Error mode, the *seterr* instruction will halt the processor.

The effect of the incorrect `Iptr` is only apparent if you subsequently switch to post-mortem debugging whereupon the debugger will complain that it is unable to locate to an `Iptr` within the kernel area. If this is a problem, you should note the `Iptr` before resuming from the breakpoint.

Setting and resuming breakpoints on an occam `STOP` statement compiled in HALT mode, will cause this problem.

8.15.18 Arrays as arguments to C functions

Because C requires a declaration of a parameter as *array of type* to be adjusted to *pointer to type* the debugger must treat all array parameters as pointers. This means that it cannot automatically display the contents of an array passed as a parameter.

In order to display the contents of arrays you should use specify the range of the array to be displayed. This is illustrated in the following example.

```
void foo (int p[4]) {
        debug_stop ();
}
```

The argument `p` will be treated as a pointer to `int` rather than an array of `int` by the C compiler. Using the INSPECT function on `p` will cause the *address* of `p` to be displayed. In order to see the *contents* of the array, the inspect command should be given an array range, for example: `p[0;3]`.

8.15.19 Backtracing with concurrent C processes

`idebug` supports backtracing from a parallel process to the parent process (where the parallel process was started via a C library call). However, for processes started asynchronously via `ProcRun`, `ProcRunHigh`, or `ProcRunLow`, `idebug` merely enables you to backtrace and does not allow operations such as inspection of variables after a backtrace. This is because the parent process which started the asynchronous processes may no longer exist, in which case inspection is meaningless.

8.15.20 Errors generated by the full C library

Generally, the full C runtime library is able to detect when there is insufficient memory for it to function correctly; in such instances it displays an error message at startup.

In rare circumstances the library is able to detect that there is insufficient memory but it does not have enough memory to display the startup error message. In such instances, it sets the error flag and terminates execution. If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger then it is highly likely that insufficient memory is available for either the static or the heap area:

```
Error: Not compiled with debugging enabled "libc.lib"
```

8.15.21 Errors generated by the reduced C library

Because the reduced C runtime library has no host to communicate with, if a run-time error occurs the reason for the error is not readily apparent. If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger then it is highly likely that insufficient memory is available for either the static or the heap area:

```
Error: Not compiled with debugging enabled "libcred.lib"
```

8.15.22 Shifting by large or negative values

The shift instructions on current transputers take time proportional to the number of places shifted — as this number is unsigned, negative values will be treated as large positive values. Large shifts will cause current transputers to temporarily 'lock' for a number of cycles equal to the number of places shifted — on 32 bit transputers this can cause the device to hang-up for up to 2^{32} cycles (approximately $3^1/_2$ minutes for a 20 MHz device).

Some languages, such as C, performs no runtime checks for invalid shift values and so do not protect you against their consequences. Other languages, such as occam, do perform such checks.

If the debugger, in post-mortem mode, locates to a source line containing a shift operator and the error flag has not been set then it is likely that a shift by a large value is taking place — this can be verified by using the INSPECT key to check the shift count.

8.15.23 C compiler optimizations

The INMOS compilers perform some code optimizations. If an external variable is optimized out from a module because it is never used then the debugger is informed of this and is able to relay the information to the user.

However, for some optimizations the debugger is not informed and consequently it may provide misleading information. The following code illustrates this:

```
int main (void){

     int     a = 0;
     int     b = 0;

     while (1) { /*  or  'for (;;)'  */
          ;
     }

     /* following code optimized out by compiler
      * as it can never be reached
      */
     a = 42;
     b = a + 1;
     a = b * b
     ...
}
```

In these cases the debugger may show the discrepancy in either of the following ways:

1 If a function follows the optimized code, the debugger associates the address of the optimized lines with the address of the start of the function.

2 If no function follows the optimized code then the debugger indicates that it is unable to find the address for any of the optimized lines.

8.16 C debugging example

This example illustrates some of the post-mortem and breakpoint features of the debugger. The debugger is run in interactive mode.

8.16.1 The example program

The example program `facs.c` calculates the sum of the squares of the first *n* factorials, using a rather inefficient algorithm. It has been structured this way for clarity in process structure and to demonstrate parallel processing and debugging methods. The same program coded in occam is supplied with the occam 2 toolset. The program incorporates five processes, each coded as a separate function. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result. The program is listed below.

```
/*****************************************
*
*  Debugger example:  facs.c
*
*  idebug (and parallel C) example based on similar program
*  in occam toolset.
*
*  Uses 5 processes to compute the sum of the squares of the
*  first N factorials using a rather inefficient algorithm.
*
*  Plumbing:
*
*  - > feed -> facs -> square -> sum -> control <--> User I/O
*  |                                         |
*   -----------------------------------------
*
*****************************************/


#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>


const double    stop_real    = -1.0;
const int       stop_integer = -1;
```

```
/*  output a double down a channel  */
void
ChanOutDouble (Channel *out, double value)

{
        ChanOut (out, (void *) &value, sizeof (value));
}


/*  input a double from a channel  */
double
ChanInDouble (Channel *in)

{
        double  value;

        ChanIn (in, (void *) &value, sizeof (value));
        return value;
}


/*  compute factorial  */
double
factorial (int n)

{
        double  result;
        int     i;

        result = 1.0;
        for (i = 1; i <= n; ++i)   {
                result = result * i;
        }
        return result;
}


/*  source stream of ints  */
void
feed (Process *p, Channel *in, Channel *out)

{
        int     n, i;

        (void) p;       /*  stop compiler usage warning  */

        n = ChanInInt (in);
        for (i = 0; i < n; ++i)   {
                ChanOutInt (out, i);
        }
        ChanOutInt (out, stop_integer);
}
```

```
/*  generate stream of factorials  */
void
facs (Process *p, Channel *in, Channel *out)

{
        int     x;
        double  fac;

        (void) p;       /*  stop compiler usage warning  */

        x = ChanInInt (in);
        while (x != stop_integer)   {
                fac = factorial (x);
                ChanOutDouble (out, fac);
                x = ChanInInt (in);
        }
        ChanOutDouble (out, stop_real);
}

/*  generate stream of squares  */
void
square (Process *p, Channel *in, Channel *out)

{
        double  x, sq;

        (void) p;       /*  stop compiler usage warning  */

        x = ChanInDouble (in);
        while (x != stop_real)   {
                sq = x * x;
                ChanOutDouble (out, sq);
                x = ChanInDouble (in);
        }
        ChanOutDouble (out, stop_real);
}
```

```
/*  sum input  */
void
sum (Process *p, Channel *in, Channel *out)

{
        double  total, x;

        (void) p;       /*  stop compiler usage warning  */

        total = 0.0;
        x = ChanInDouble (in);
        while (x != stop_real)   {
                total = total + x;
                x = ChanInDouble (in);
        }
        ChanOutDouble (out, total);
}

/*  user interface and control  */
void
control (Process *p, Channel *in, Channel *out)

{
        double  value;
        int     n;

        (void) p;       /*  stop compiler usage warning  */

        printf ("Sum of the first n squares of factorials\n")
;
        do   {
                printf ("Please type n : ");
        }  while (scanf ("%d", &n) != 1);
        printf ("n = %d\n", n);
        printf ("Calculating factorials ... ");

        ChanOutInt (out, n);
        value = ChanInDouble (in);

        printf ("\nThe result was : %g\n", value);
}
```

```
Channel *
Checked_ChanAlloc ()

{
        Channel *chan;

        if ((chan = ChanAlloc ()) == NULL)    {
                fprintf (stderr, "ChanAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return chan;
}

Process *
Checked_ProcAlloc (void (*func)(), int sp, int nparam,
                      Channel *c1, Channel *c2)

{
        Process *proc;

        proc = ProcAlloc (func, sp, nparam, c1, c2);
        if (proc == NULL)    {
                fprintf (stderr, "ProcAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return proc;
}
```

```
int
main (void)

{
        Channel *facs_to_square, *square_to_sum;
        Channel *sum_to_control, *feed_to_facs;
        Channel *control_to_feed;

        Process *p_feed, *p_facs, *p_square;
        Process *p_sum, *p_control;

        facs_to_square  = Checked_ChanAlloc ();
        square_to_sum   = Checked_ChanAlloc ();
        sum_to_control  = Checked_ChanAlloc ();
        feed_to_facs    = Checked_ChanAlloc ();
        control_to_feed = Checked_ChanAlloc ();

        p_feed = Checked_ProcAlloc (feed, 0, 2,
                         control_to_feed, feed_to_facs);
        p_facs = Checked_ProcAlloc (facs, 0, 2,
                         feed_to_facs, facs_to_square);
        p_square = Checked_ProcAlloc (square, 0, 2,
                         facs_to_square, square_to_sum);
        p_sum = Checked_ProcAlloc (sum, 0, 2,
                         square_to_sum, sum_to_control);
        p_control = Checked_ProcAlloc (control, 0, 2,
                         sum_to_control, control_to_feed);

        ProcPar (p_feed, p_facs, p_square, p_sum,
                                         p_control, NULL);

        exit (EXIT_SUCCESS);
}
```

8.16.2 Compiling and loading the example

The source of the program is provided in the toolset `debugger` examples subdirectory. It should be compiled for transputer class TA with debugging enabled, then linked with the appropriate library files and made bootable using `icollect` with the '`T`' option to create single transputer bootable code.

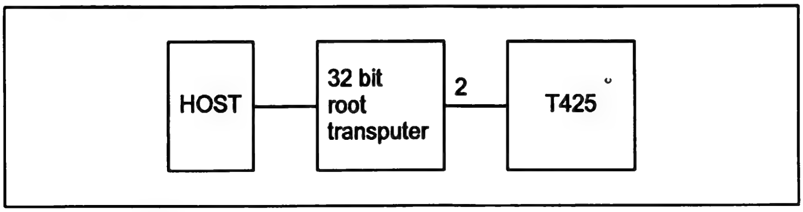


Figure 8.4 Hardware configuration for the example

The example is intended for running on a B008 board wired *subs*. See section 4.7 in the *Toolset Reference Manual* debugger chapter if your system is different.

A typical sequence of commands for compiling, linking, and booting the program is shown below. The 'I' option on the linker command line is optional but provides useful information on the progress of the linking operation.

Command sequences are shown for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

UNIX:

```
icc facs.c -g -ta -o facs.tax
ilink facs.tax -f cnonconf.lnk -ta -o facs.cah -i
icollect facs.cah -t
```

MS-DOS/VMS:

```
icc facs.c /g /ta /o facs.tax
ilink facs.tax /f cnonconf.lnk /ta /o facs.cah /i
icollect facs.cah /t
```

The program is loaded for breakpoint debugging by running `idebug` with in interactive mode using one of the following commands:

```
idebug -sr -si -b2 facs.btl -c t425

idebug /sr /si /b2 facs.btl /c t425
```

This command starts up the debugger and displays the Monitor page but does not start the program. The `iserver` 'SI' switch is optional.

Note: If your transputer is not a T425 you should change the `t425` option to the appropriate transputer type. You may also need to change the number specified after the 'B' option to the number of the root transputer link to which the network is connected. See table 4.4 in chapter 4 of the *Toolset Reference Manual* for more details about the options to use, if in doubt.

8.16.3 Setting initial breakpoints

Initial breakpoints can often be set by using the Monitor page $\boxed{B}$ command and specifying a breakpoint at the start of `main()`. In this example we use a different method based on setting specific breakpoints in the source code before the program is started.

At the Monitor page select $\boxed{F}$ to display the source file. At the object module filename prompt specify the compiled object file `facs.tax`. The debugger uses debug information within the object module to select the source file.

The source file is displayed with the cursor positioned at the first function definition. At this point the program is still waiting to be started.

Set a breakpoint at the beginning of the `ChanOutDouble()` function using TOGGLE BREAK. The debugger confirms the breakpoint is set and gives the breakpoint a unique identification number (note that the breakpoint is set on the first executable line of the function).

8.16.4 Starting the program

Return to the Monitor page using the MONITOR key and start the program by selecting the J command. Press RETURN at the '`Command line`' prompt (no command line is required) and give a small positive number (e.g. 12) when the program prompts for input. The program runs until it reaches the breakpoint.

8.16.5 Entering the debugger

At the breakpoint the debugger displays the number of the breakpoint and the number of times it has been encountered (or hit) and then requests confirmation to continue the stopped process. Press any key except R or r to enter the symbolic debugging environment. The debugger locates to the breakpoint and displays the source code.

8.16.6 Inspecting variables

Variables and channels in `ChanOutDouble()` can now be examined. For example, to examine the variable `value` press INSPECT and specify its name at the prompt. The debugger displays the value 1.0 and labels it as a `double`. Pressing INSPECT with the cursor positioned on `value` has the same effect.

Note that only variables in scope at the debugger's current location point can be inspected, although the rest of the file can be displayed with the cursor keys. The current location point is at the start of function `ChanOutDouble()`.

8.16.7 Finding addresses of variables

The debugger provides a comprehensive C expression language which may be used with INSPECT and MODIFY. To obtain the address of a variable, you use the same expression as you would in a C program. Press INSPECT and specify `&value` to display the address of `value`. Notice that addresses are displayed in hex notation by default. TOGGLE HEX may be used to display the values of variables in hex notation if required.

8.16.8 Backtracing

`ChanOutDouble()` is called from function `facs()` to output the factorial it calculates for each integer received from `feed()`. To confirm this press BACKTRACE and the debugger locates to the line in `facs()` where `ChanOutDouble()` is called. Press $\boxed{\text{TOP}}$ to return to where the breakpoint occurred. Now press $\boxed{\text{TOGGLE BREAK}}$ to remove the breakpoint on this line.

8.16.9 Jumping down a channel

Within `facs()` the variable `fac` is the first in a sequence of outputs on the channel `out`. To trace the destination process for `fac` first use $\boxed{\text{INSPECT}}$ to see the value of the channel `out`, which is declared to be a channel pointer. Use $\boxed{\text{INSPECT}}$ again but this time specify `*out`, which de-references the channel pointer. The debugger displays an `Iptr` and `Wdesc`, indicating that there is a low priority process waiting at the other end of the channel.

Now press $\boxed{\text{CHANNEL}}$ and again specify `*out` to de-reference the channel pointer. The debugger jumps down the channel connecting the two processes and locates to `ChanInDouble()`. Now backtrace to the function which called `ChanInDouble()` to input a value, namely function `square()`. Variables in scope now become available for inspection (at this stage they have not been initialized).

While still in function `square()` move the cursor to the first line containing `ChanOutDouble()` and set a breakpoint. Then press $\boxed{\text{RESUME}}$ in order to run the program up to the breakpoint just set.

8.16.10 Inspecting by expression

In function `square()` inspect the variable `sq` and check the computation by $\boxed{\text{INSPECT}}$ and specifying the expression `x * x`. Note how $\boxed{\text{INSPECT}}$ can be used to perform arithmetic on any variable in scope. Expressions can also include numbers and other variables and constants in scope at the location point.

Press $\boxed{\text{INSPECT}}$ and type `x != stop_real` in order to see the value used to control the while loop.

8.16.11 Modifying a variable

In breakpoint debugging any program variable may be modified. To modify a variable `x` press $\boxed{\text{MODIFY}}$ and specify `x` at the '`Destination`' prompt. The debugger now requests the new value by display the '`Source`' prompt. Enter any value and check the value has changed by inspecting `x` once again.

8.16.12 Backtracing to `main()`

While still in `square()`, press $\boxed{\text{BACKTRACE}}$ to locate back to where the function was called. The debugger locates to `ProcPar()` in function `main()` where the five major processes are started in parallel. If the call to function `square()` had been nested in other calls, successive BACKTRACE operations might have been necessary but would have eventually located to the call in the program main function.

8.16.13 Entering `#include` files

Press GOTO LINE and select line 20. This will locate you to the line `#include <stdio.h>`. By using the ENTER FILE key you may now enter the `#include` file (and then any nested files within it); the EXIT FILE key will bring you out again into the enclosing file.

8.16.14 Quitting the debugger

Finally, to quit the debugger use the FINISH key (you may also quit the debugger from the Monitor page using the Q command). If the debugger was run with the '`XQ`' option, then it will prompt for confirmation before exiting.

8.17 occam debugging example

This example illustrates some of the post-mortem and breakpoint features of the debugger. The debugger is run in interactive mode.

8.17.1 The example program

The example program `facs.occ` calculates the sum of the squares of the first *n* factorials, using a rather inefficient algorithm. It has been structured this way for clarity in process structure and to demonstrate parallel processing and debugging methods. The same program coded in C is supplied with the C toolset. The program incorporates five processes, each coded as a separate procedure. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result. The program is listed below.

Note: Triple braces ({{{ and }}}) in the listing indicate *fold* marks in the program. They are retained for compatibility with the folding editors often used for writing occam programs.

```
-----------------------------
--
--  Debugger example:  facs.occ
--
--  Uses 5 processes to compute the sum of the squares
--  of the first N factorials using a rather inefficient
--  algorithm.
--
--  Plumbing:
--
--   feed-> facs-> square-> sum-> control <--> User IO
--     |                             |
--     ------------------------------
--
-----------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"

PROC facs.entry (CHAN OF SP fs, ts, []INT free.memory)

  VAL stop.real    IS -1.0 (REAL64) :
  VAL stop.integer IS -1 :

  --{{{  FUNC factorial  -  compute factorial
  REAL64 FUNCTION factorial (VAL INT n)
    REAL64 result :
    VALOF
      SEQ
        result := 1.0 (REAL64)
        SEQ i = 1 FOR n
          result := result * (REAL64 ROUND i)
      RESULT result
  :
  --}}}

  --{{{  PROC feed       -  source stream of integers
  PROC feed (CHAN OF INT in, out)
    INT n :
    SEQ
      in ? n
      SEQ i = 0 FOR n
        out ! i
      out ! stop.integer
  :
  --}}}
```

```
--{{{  PROC facs       -  generate stream of factorials
PROC facs (CHAN OF INT in, CHAN OF REAL64 out)
  INT x :
  REAL64 fac :
  SEQ
    in ? x
    WHILE x <> stop.integer
      SEQ
        fac := factorial (x)
        out ! fac
        in ? x
    out ! stop.real
:
--}}}

--{{{  PROC square      -  generate stream of squares
PROC square (CHAN OF REAL64 in, out)
  REAL64 x, sq :
  SEQ
    in ? x
    WHILE x <> stop.real
      SEQ
        sq := x * x
        out ! sq
        in ? x
    out ! stop.real
:
--}}}

--{{{  PROC sum         -  sum input
PROC sum (CHAN OF REAL64 in, out)
  REAL64 total, x :
  SEQ
    total := 0.0 (REAL64)
    in ? x
    WHILE x <> stop.real
      SEQ
        total := total + x
        in ? x
    out ! total
:
--}}}
```

```
  --{{{  PROC control    -  user interface and control
  PROC control (CHAN OF SP fs, ts,
                CHAN OF REAL64 result.in,
                CHAN OF INT n.out)
    REAL64 value :
    INT n :
    BOOL error :
    SEQ
      so.write.string.nl (fs, ts,
              Sum of the first n squares of factorials")

      error := TRUE
      WHILE error
        SEQ
          so.write.string (fs, ts, "Please type n: ")
          so.read.echo.int (fs, ts, n, error)
          so.write.nl (fs, ts)

      so.write.string(fs, ts, "Calculating factorials...")

      n.out ! n
      result.in ? value

      so.write.nl (fs, ts)
      so.write.string (fs, ts, "The result was: ")
      so.write.real64 (fs, ts, value, 0, 0) --  free format
      so.write.nl (fs, ts)
      so.exit (fs, ts, sps.success)
  :
  --}}}

  CHAN OF REAL64 facs.to.square, square.to.sum :
  CHAN OF REAL64 sum.to.control :
  CHAN OF INT feed.to.facs, control.to.feed :

  PAR
    feed (control.to.feed, feed.to.facs)
    facs (feed.to.facs, facs.to.square)
    square (facs.to.square, square.to.sum)
    sum (square.to.sum, sum.to.control)
    control (fs, ts, sum.to.control, control.to.feed)
:
```

8.17.2 Compiling the `facs` program

The source of the program is provided in the toolset examples subdirectory. It should be compiled for transputer class TA with debugging enabled, then linked with the appropriate library files and made bootable using `icollect` with the 'T' option to create single transputer bootable code. The example is intended for running on a B008 board wired *subs*. See section 4.7 in the *Toolset Reference Manual* debugger chapter if your system is different.

Using `imakef`

If your system has a *make* utility you may use **imakef** to generate a suitable makefile to help build the program:

```
imakef facs.bah

make -f facs.mak                (UNIX)
make /f facs.mak                 (MS-DOS/VMS)
```

Using the tools directly

A typical sequence of commands for compiling, linking, and booting the program is shown below. The 'I' option on the linker command line is optional but does provide useful information on the progress of the linking operation.

Command sequences follow for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

UNIX:

```
oc -ta facs.occ -o facs.tah
ilink -ta facs.tah hostio.lib convert.lib -f occama.lnk
      -o facs.cah
icollect -t facs.cah -o facs.bah
```

MS-DOS/VMS:

```
oc /ta facs.occ /o facs.tah
ilink /ta facs.tah hostio.lib convert.lib /f occama.lnk
      /o facs.cah
icollect /t facs.cah /o facs.bah
```

8.18 Breakpoint debugging

The following section demonstrates how to debug the example program in interactive mode.

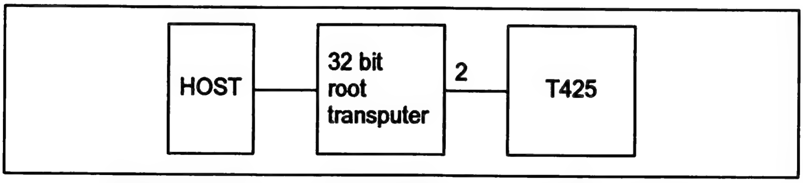


Figure 8.5 Hardware configuration for breakpoint example

8.18.1 Loading the program

The program is loaded for breakpoint debugging by running **idebug** in interactive mode using one of the commands given below. Use the appropriate command for your system.

```
idebug -sr -si -b2 facs.bah -c t425  (UNIX)

idebug /sr /si /b2 facs.bah /c t425  (MS-DOS/VMS)
```

This command starts up the debugger and displays the Monitor page but does not start the program. The `iserver` '`SI`' switch is optional.

Note: If your transputer is not a T425 you should change the `t425` option to the appropriate transputer type. You may also need to change the number specified after the '`B`' option to the number of the root transputer link where your network is connected. See table 4.4 in chapter 4 of the *Toolset Reference Manual* for more details about the options to use if in doubt.

8.18.2 Setting initial breakpoints

Initial breakpoints can often be set with the Monitor page $\boxed{\text{B}}$ command and specifying an entry point breakpoint (this would set a breakpoint at `facs.entry`). In this example a different method is used based on setting specific breakpoints in the source code before the program is started.

At the Monitor page select option $\boxed{\text{F}}$ to display the source file. At the object module filename prompt specify the compiled object file `facs.tah`. The debugger uses debug information within the object module to select the source file. The source file `facs.occ` is displayed with the cursor positioned at the first procedure definition, namely `facs.entry`. At this point the program is still waiting to be started.

Use $\boxed{\text{GOTO LINE}}$ to move the cursor to line 56 (`out ! fac`) and set a breakpoint there using $\boxed{\text{TOGGLE BREAK}}$. The debugger confirms the breakpoint is set and gives the breakpoint a unique identification number.

8.18.3 Starting the program

Return to the Monitor page using the $\boxed{\text{MONITOR}}$ key and start the program by selecting the $\boxed{\text{J}}$ command. Press $\boxed{\text{RETURN}}$ at the '`Command line`' prompt (no command line is required) and give a small positive number (e.g. 12) when the program prompts for input. The program runs until it reaches the breakpoint.

8.18.4 Entering the debugger

At the breakpoint the debugger displays the number of the breakpoint and the number of times it has been encountered (or hit) and then requests confirmation to continue the stopped process. Press any key except $\boxed{\text{R}}$ or $\boxed{\text{r}}$ to enter the symbolic debugging environment. The debugger locates to the breakpoint and displays the source code.

8.18.5 Inspecting variables

Variables and channels in `facs` can now be examined. For example, to examine the variable `fac` move the cursor to `fac` and press $\boxed{\text{INSPECT}}$. The debugger displays the value as `REAL64 1.0` and gives its address. Pressing INSPECT with the cursor positioned on a space causes the debugger to prompt you for a symbol. Note that only variables in scope at the debugger's current location point can be inspected, although the rest of the file can be displayed with the cursor keys. The current location point is line 56 in the procedure `facs`.

8.18.6 Backtracing

`facs` is called in parallel by `facs.entry` to output the factorial it calculates for each integer received from `feed`. To confirm this press BACKTRACE and the debugger locates to the line in `facs.entry` where `facs` is called. Press TOP to return to where the breakpoint occurred. The current location point is line 56 in the procedure `facs`.

8.18.7 Jumping down a channel

Within `facs` the variable `fac` is the first in a sequence of outputs on the channel `out`. To trace the destination process for `fac` first INSPECT the channel `out`. The debugger displays an `Iptr` and `Wdesc`, indicating that there is a low priority process waiting at the other end of the channel.

Now press CHANNEL and again specify `out`. The debugger jumps down the channel connecting the two processes and locates to the corresponding channel input in procedure `square` (the statement `in ? x`). Variables in scope within `square` now become available for inspection (at this stage they have not been initialized).

8.18.8 Modifying a variable

In breakpoint debugging program variables may be modified. Start by first inspecting `x` in order to ensure that the new value will be different. To modify the variable `x` position the cursor on `x` and press MODIFY. At the modify value prompt specify the value to be placed in `x`. Note that the modify prompt reminds you of the type of `x`. Enter any valid value and check the value has changed by inspecting `x` once again.

8.18.9 Entering `#INCLUDE` files

Press GOTO LINE and select line 17. This will locate you to the line `#INCLUDE "hostio.inc"`. By using the ENTER FILE key you may now enter the `#INCLUDE` file (and any then nested files within it); the EXIT FILE key will bring you out again into the enclosing file.

8.18.10 Resuming the program

To resume execution of the program from the current breakpoint press the RESUME key. This will cause the program to continue running until it encounters the breakpoint again. Press an appropriate key to enter the symbolic debugging environment. This will cause the debugger to locate to line 56.

8.18.11 Clearing a breakpoint

To clear the breakpoint already set at line 56 use the TOGGLE BREAK key. The debugger will confirm that the breakpoint has been cleared. Press RESUME to resume execution and cause the program to display its result. The debugger will confirm that the program has finished and will pause in order to enable you to read the output from the program. Press any key as indicated to enter the Monitor page. Note that the Monitor page displays the exit status from the program.

8.18.12 Quitting the debugger

Finally, to quit the debugger you can use the Monitor page Q command. You may also quit the debugger from symbolic mode by using the FINISH key. If the debugger was run with the 'XQ' option, then it will prompt for confirmation before exiting.

8.19 Post-mortem debugging

The following section demonstrates how to debug the example `facs` program in post-mortem mode.

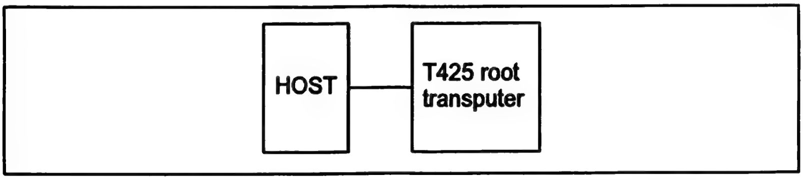


Figure 8.6 Hardware configuration for post-mortem example

8.19.1 Running the example program

When you have built an executable code file you can run the program by typing one of the following commands:

```
iserver -se -sb facs.bah      (UNIX)
iserver /se /sb facs.bah      (MS-DOS/VMS)
```

The program immediately prompts you for a value. For correct execution the number must be less than 100. To create an error for the purpose of this example, enter the value 101 and press RETURN. The program will fail with the message: `Error - iserver - Error flag raised by transputer.`

8.19.2 Creating a memory dump file

To create a memory dump file for the debugger to read, type:

```
idump facs 15000
```

This creates a file called `facs.dmp` containing the transputer's register contents and the first 15000 bytes of memory. You are then returned to the operating system prompt.

8.19.3 Running the debugger

To debug the example program, use one of the following commands:

```
idebug -si facs.bah -r facs -c t425 (UNIX)
idebug /si facs.bah /r facs /c t425 (MS-DOS/VMS)
```

The `iserver` '`SI`' switch is optional. The '`R`' option identifies the program as one that was executed on the *root* transputer and specifies the memory dump file to be read.

Note: If your transputer is not a T425 you should change the `t425` option to the appropriate transputer type.

Should you wish to run the debugger a second time on this single processor example, without an intervening `idump` command, you will need to add the `iserver` '`SR`' option to the command line to reset the network.

The debugger first displays its version number, then some processing information, and eventually locates to the source line from which the error was generated:

```
sq := x * x
```

You can now begin to debug the program. You can use the symbolic facilities to browse the source, locate to specific lines and areas of code, inspect variables and channels, and trace procedure calls, and you can inspect and disassemble memory using the Monitor page commands.

The following sections illustrate some of the debugging operations you can perform on the example program. For further details about any of the debugging functions described in these sections, see chapter 4 of the *Toolset Reference Manual*.

Inspecting variables

When the debugger is displaying source code, you may inspect any variable by placing the cursor on the variable and pressing INSPECT.

For example, to display the value of `x`, place the cursor over `x` in the source code and press INSPECT. `x` is displayed in both decimal and hexadecimal forms, and its address in memory is given in hexadecimal. For example:

```
REAL64 'x' has value ...
9.3326215443944096E+155 (#605166C698CF1838) (at #80000464)
```

In the same way you can inspect the values of `sq`, `square`, `stop.integer`, `stop.real`, and any other variable or constant that is in scope. Use the cursor keys to scroll through the code. To return to the source of the original error, use the RELOCATE function. You can also use the INSPECT function to examine procedures and functions. If you place the cursor on a procedure or function name and press INSPECT, the debugger displays its address and workspace requirements. You can also examine any symbol in the source by specifying its name. To do this, move the cursor to a blank area and press INSPECT. The debugger then prompts for the symbol name.

Inspecting channels

The debugger can also examine processes on channels within the scope of the original error. If you place the cursor on channel `out` and press INSPECT, information about the channel is displayed. For example:

```
CHAN 'out' has Iptr:#800022F8 and Wdesc:#80000381 (Lo) (at
#8000063C)
```

This indicates that there is a process waiting for communication on channel `out`, and that it is a low priority process. To find out which occam process is waiting, press CHANNEL. The cursor will be placed on the line corresponding to the other process, which in this example is inside the procedure `sum`, on the following line:

```
    in ? x
```

Within procedure `sum`, you can examine any symbol using INSPECT. Within the `sum` procedure you can inspect the channel `out` and use CHANNEL to jump to the waiting process, which is the procedure `control` that is waiting for the final result. Again you can use INSPECT to examine any symbol.

Retracing and Backtracing

So far the debugger has located three of the five processes that compose the program. What about the others? First use the RETRACE key to retrace your steps back to procedure `square`. When in procedure `square`, inspect channel `in`, which is connected to the `facs` procedure. It is empty, which means that no process is waiting to communicate.

Next try BACKTRACE. This function backtraces down nested procedure calls. Each time the function is used the debugger locates to the line in the enclosing code from which the procedure was called.

In this example, BACKTRACE moves the cursor to the line where procedure `square` is called. Again, you can inspect any symbol which is in scope at this line. For example, you can inspect the channels `feed.to.facs` and `facs.to.square`. Both should be empty, which means that the remaining processes were actively executing, rather than waiting to communicate, when the program halted.

To find the active processes, you need to examine the transputer's process queues using the Monitor page facilities, as described below.

Displaying process queues

To display the process queues, first enter the debugger Monitor page from the symbolic environment by pressing the MONITOR key. Low level information is displayed for the current processor, along with a list of Monitor page commands.

To display the process queues, use the Monitor page R command. This displays two active processes, identified by their respective `Iptr` and `Wdesc`. When you have identified the processes to examine, you can use the Monitor page G command to jump to those processes and inspect the code. Other commands to try from the Monitor page are T, which displays the processes waiting on the transputer's timers; and L, which displays processes waiting for communication on the transputer's links.

Goto process

When you press G, the following message is displayed:

```
[CURSOR] then [RETURN], or 0 to F, (I)ptr, (L)o, or (Q)uit
```

To display the first active process[1], type 0 (zero). The cursor will be placed on the following source line (in procedure '`feed`'):

```
      out ! i
```

Because this process is on the queue and not waiting, it must have already performed the communication and is about to resume executing. You can examine variables within the procedure as before.

To display the last remaining process in the program, press MONITOR again, and type G followed by 1 to locate to the second process in the queue. This process will either be executing code within the compiler libraries or within the replicated `SEQ`. If it is executing code within a library, the debugger displays the call to the library routine rather than the source itself, because the source is not supplied. For example:

```
      result := result * (REAL64 ROUND i)
```

Again, you may inspect variables within the process. For example, by inspecting the variable '`i`', you can determine how many times the loop has been executed. Or you can use BACKTRACE to determine where the function was called from.

1. For a full explanation of the possible responses see the definition of the Goto Process command in the `idebug` reference chapter (chapter 4 of the *Toolset Reference Manual*).

Advanced techniques

9 Advanced use of the configurer

This chapter describes the advanced use of `icconf` and is aimed at users who wish to override certain configuration defaults. The chapter deals with two topics:

- Memory usage by the configurer.
- Channel communications.

The chapter describes how to override the default allocation of user's code and data in memory and how to refine the channel communication for the target network using advanced virtual routing techniques. An example configuration using virtual routing is provided at the end of the chapter.

9.1 Code and data placement

The configuration language provides one processor attribute (`reserved`) and two classes of process attributes (`location` and `order`), for influencing the use of memory. The syntax of these attributes is described in sections 6.1.2 and 6.1.3 respectively. This section describes the circumstances in which the attributes should be used.

Note: when the `location` or `order` attributes are used, debugging using the toolset debugger `idebug` is not supported.

9.1.1 Default memory map

By default the configurer maps code and data into memory in the following order beginning at LoadStart: stack; code; vector space; static; heap and system data. The memory segments are contiguous. The upper limit of the memory available to the configurer is defined by the `memory` attribute specified for the processor nodes.

By default, the configurer only knows about this continuous block of memory, whose upper and lower limits are set by the value of `memory` minus the LoadStart offset for the processor. The default memory map is illustrated in Figure 9.1.

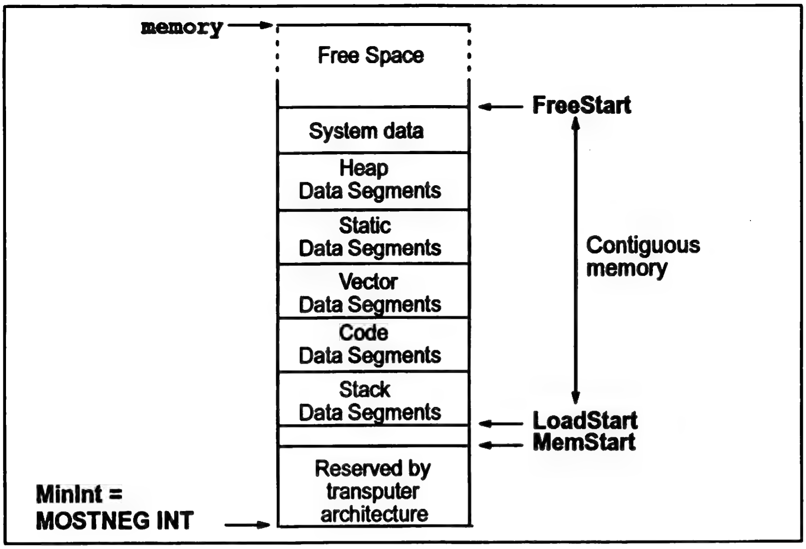


Figure 9.1 **icconf** default memory map

The first 2 or 4 Kbytes of memory above **MOSTNEG INT** is implemented as on–chip RAM, and includes a few words which are reserved by the transputer hardware for the implementation of links and other hardware registers.

LoadStart is either just above or coincident with **MemStart**, see section 2.3.11 of the *ANSI C Toolset Reference Manual.*

9.1.2 Other memory configurations

Figure 9.2 illustrates a memory configuration with additional requirements to those provided by the configurer in default mode. To cater for such situations the `reserved` and `location` attributes are supported by the configuration language.

Figure 9.2 illustrates two different sets of possible requirements:

- The first is where the available memory is discontinuous and the lowest block of memory is not sufficiently large enough to hold all the code and data.

- The second is where a block of memory is available outside the default range of memory addressed by the configurer, (see above).

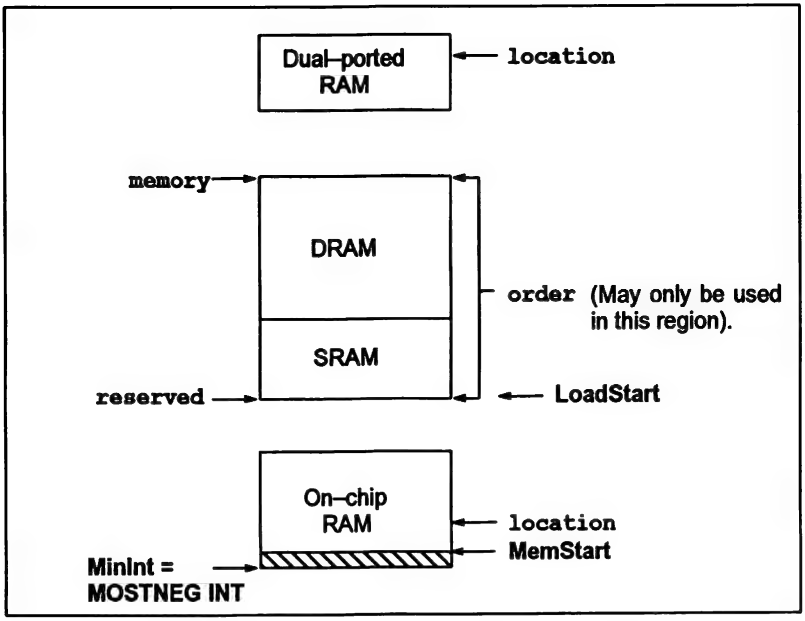


Figure 9.2 Example discontinuous memory map

9.1.3 `reserved` processor attribute

This attribute is used to specify the size of memory, in bytes, to reserve from **MOSTNEG INT** which cannot be used by the configurer to place user and system processes. It will be possible for the user, using the `location` attributes to place the code and data segments of user processes into the reserved memory.

Checks are performed to ensure that the `reserved` memory size is greater than the default **LoadStart** offset for the processor and less than the memory size specified by the `memory` attribute. The configurer will also ensure that the size is word aligned by rounding the size up to the nearest word boundary. **Note:** the value of the default **LoadStart** is variable, see section 2.3.11 of the *ANSI C Toolset Reference Manual.*

Example:

processor `(reserved = 5K);`

In figure 9.2 the `reserved` attribute has been used to force the configurer to place system and user code into the second block of memory and to ignore the on–chip RAM.

When the `reserved` attribute is used, the region of memory available to the configurer for automatically placing the non–addressed code and data segments of processes is defined as being:

> the top of memory as specified by the `memory` attribute minus the memory size specified by the `reserved` attribute.

If no `reserved` attribute is defined then the region of memory available to the configurer is:

> the top of memory as specified by the `memory` attribute minus the default **LoadStart** offset for the processor.

9.1.4 `location` process attribute

The `location` attributes are optionally used to specify absolute addresses for the code and data segments of a process. The `location` attribute has the sub–attributes **`stack`**, **`code`**, **`vector`**, **`static`** and **`heap`** for process nodes and types. These attributes override the equivalent `order` attributes if specified.

Checks are performed to ensure that any code and data segments that have been absolutely addressed using the `location` attributes are not placed into an illegal region of memory, such as the:

- memory used by the configurer for automatically placing code and data segments i.e. the region defined by **Loadstart** and the `memory` attribute. (See section 2.3.11 of the *ANSI C Toolset Reference Manual* for more information about **Loadstart**).
- address locations that exceed the highest possible memory address location for the processor.

The configurer will fail with an error message if either of the above occur. An error will also be received if the addresses specified are not word aligned.

A further check is made that the addresses are non-overlapping and a warning will be generated if they are. It is not illegal to have overlapping regions of memory within the permitted regions for configuration code, as described above. However, it is user's responsibility to ensure there is no conflict in the use of overlapping regions at runtime.

A warning will also be generated if the `location` attributes place code or data at address locations that exist below **MemStart**.

If the `location` attributes are not specified then the configurer will automatically place non-addressed code and data segments.

Example (on a 32–bit processor):

```
process (location (code = 0x80000100)....
```

> This example specifies the start address for the process code segment. It assumes that LoadStart has been redefined, using the `reserved` attribute.

Figure 9.2 indicates how the `location` attributes can be used to access memory below **LoadStart** (which has been changed from its default value by the `reserved` attributes) or spare memory locations available on external RAM.

9.1.5 `order` process attribute

The `order` process attributes, described in section 6.1.3, can still be used in conjunction with the `reserved` and `location` attributes. The `order` attributes are used to change the ordering priority of those process segments automatically placed by the configurer i.e. non–addressed code and data segments. They only operate within the memory region delimited by **LoadStart** and the value of the `memory` attribute.

9.1.6 `location` versus `order` attribute

The `location` and `order` attributes have the same sub–attributes, namely:

`stack`, `code`, `vector`, `static` and `heap`

As stated in section 9.1.4 if both the `location` and `order` attributes are specified for a particular segment, e.g. `stack`, then the `location` attribute will override the `order` attribute.

However, if an ambiguous process declaration is made, it will be assumed that the `order` attribute is intended. For example:

```
process (code = n);
```

is the same as:

```
process (order ( code = n));
```

Section 6.2.13 gives details of the syntax of process node declarations.

9.2 Channel communication – configuration techniques

When software virtual routing is required, the configurer works by adding multiplexing and de-multiplexing processes to implement a number of virtual channels over a single hardware link. It will also add routing processes to through-route data between processors which are not directly connected. In doing so it assumes by default that:

- any link to link connections in the target network can be used for implementing virtual channel traffic.
- any of the processors can be used for through-routing.
- where multiple routes of the same length exist between two processors, the virtual channels between these processors should be shared out between these routes as much as possible.

While these are, in general, reasonable assumptions, users may require more control over how processors and links are used for implementing virtual channels in specific networks. The configurer permits users to control its routing decisions by means of processor attributes and channel placements which can be defined in the configuration source file. These are designed to supply the following capabilities:

- A channel may be placed on a specific hardware link between processors. This instructs the configurer to implement the channel directly using the hardware link rather than as a virtual channel. Only two channels may be placed (one in each direction) on a hardware link. This can be used to ensure that a limited number of critical channels are directly implemented by hardware links.
- It is possible to prevent specific processors from being used as pathways for virtual channels required by other processors. This ensures that certain critical processors within the target system are not used for through-routing virtual channels for less critical processors.
- It is possible to ensure that all virtual channels are routed via a group of processors specifically placed in the target network to support them. Hence a group of small inexpensive processors may be placed in the middle of a network of processors to provide the communications requirements at little cost to the other processors.
- It is possible to control the number of virtual channel support processes that are added to particular processors, and also whether they are given use of internal memory in preference to application processes. This preserves the performance of critical processors in the target network and allows virtual channel support on processors with limited memory capacity.

The following sections describe the use of the `place` statement and the `order` attribute to optimize important channels and to make the best use of fast memory. Section 9.2.3 introduces the additional attributes used to control the configurer's routing system and describes how to use them to meet the requirements identified above. An example is included in section 9.2.4.

9.2.1 Optimizing important application channels

By placing an internal application channel on a hardware link (at either end or both ends) it is possible to reserve the hardware link solely for the use of the application channel concerned.

With this technique a sub-set of the channels used by an application can be placed on a sub-set of the hardware links available within the target system. This then optimizes the performance of the placed data paths.

When doing the placement the user must be careful to leave at least enough free links to form a minimal spanning tree between each sub-set of processors in the target network that require through-routed virtual channels to connect them. (See section 9.2.3).

9.2.2 Virtual communications – use of fast memory

Normally the stack segment of virtual channel support processes (added to the target network by the configurer) is allocated within fast memory (i.e. at the most negative addresses) before the user process code and data segments are allocated.

User process code and data segments can, however, be allocated from internal store before the stack of the virtual channel support processes is allocated. This is done by setting `order` attributes for the relevant user processes to lower values than those automatically given to the stack segments of the virtual channel support processes.

The stack segments of virtual channel support processes placed by the configurer are all given the `order` value `-20000` or (as defined in `setconf.inc`) `ROUTER_ORDER`. The stack segments of multiplexing and de-multiplexing processes placed by the configurer are all given the order value `-10000` or `MUXER_ORDER`.

If `order` values on the code and data segments of user processes are less than `ROUTER_ORDER` the segments concerned will be allocated from internal store before any of the virtual channel support processes' stacks are allocated.

If `order` values on user processes are less than `MUXER_ORDER` and greater than `ROUTER_ORDER`, only the stack segments required by virtual channel support processes will be allocated before the configurer allocates space for the user processes concerned.

If the stack segments of heavily–used virtual channel support processes are pushed out of internal store by giving priority to user processes, the impact on the performance of the virtual links and the processor will be quite noticeable. User processes should only be given priority over the virtual channel support processes on a processor if the amount of data through-routed by the processor during normal operation is likely to be small.

Giving user processes priority use of fast memory will only impact the performance of those virtual channels used by processes on the processor. The CPU cost of supporting those virtual channels will only be slightly increased.

9.2.3 Control of routing and placement

This section describes how the allocation of a virtual routing system across a network can be controlled. For example, particular routes can be avoided or promoted as required.

Introduction to routing and placement attributes

User control of routing and placement is done by means of an extra processor attribute, which has three sub–attributes and is specified using the configuration language. The syntax of the attributes is as follows:

```
processor (router (routecost = exp));
processor (router (tolerance = exp));
processor (router (linkquota = exp));
```

The `router` attribute introduces the sub–attributes which influence different aspects of the routing algorithm. These sub–attributes are described, in turn, below.

Routing cost

The first of the attributes – `routecost` – can be used to make the configurer choose one processor over another when deciding how to route channels in the network. In the default case, all processors and links in the network are assumed to be equally usable. When deciding how to route a channel between two processors, the configurer works out the routes between the two points, and then calculates the “cost” of each route by counting the number of processors on each route. The “best” of these (the one with the least number of processors) is then chosen to implement the channel, and the appropriate through-routing processes are placed on each intermediate processor on the route. If there are a number of channels to be implemented between the two ends, and there is more than one route of the same (“best”) length available, then the channels are shared between the available routes.

The `routecost` attribute allows a *routing cost* to be explicitly allocated to one or more processors in the network. The cost of a route between two processors is then determined not simply by the number of intermediate processors, but by the *sum of the routing costs* of all the intermediate processors. There is a default routing cost for processors which have not had one explicitly allocated. So by giving a *high* routing cost value to a processor, this will discourage the configurer from using it as an intermediate node when routing channels. Similarly by giving it a *low* cost compared with other processors in the network, this will encourage the configurer to use it for through-routing.

Tolerance

The second attribute – `tolerance` – controls how the configurer decides to share out channels between available routes. If there are a number of channels to be implemented between two processors, then the configurer normally calculates the cost of each possible route, and then shares out the channels between available “best” routes with the least cost. If there is only one ''best'' route then all the channels will go via that one. In some circumstances it may be better to share out the channels more evenly, to prevent bottlenecks in the system, even if this results in some channels being implemented on slightly higher cost routes. The `tolerance` attribute for a processor is designed to allow this.

When calculating whether to use a route for channel sharing, the configurer uses the minimum of the `tolerance` values of the processors on that route. It subtracts that tolerance from the route cost; if the result is less than the cost of the "best" route, then this route, as well as the "best" routes, may be used for load-sharing of channels. As an example, consider a network in which all processors have been given the same routing cost (say 1000). Normally, this would result in load-sharing of channels only when the routes are the same length. However, if the tolerance of all the processors were set to twice the routing cost value (2000), then the configurer would also include routes with one more processor on them than the "best" route for channel load-sharing.

When setting up a network, the `routecost` attributes should be set first to indicate which processors are preferred for through-routing. Then the `tolerance` attribute can be set, for all processors in the network, to influence the load-sharing strategy. In general a set of processors in a network (or in part of a network) would be given the same `tolerance` value to indicate the load-sharing strategy required for that network (or part of the network). The likely cases are:

- A zero `tolerance` value indicates that virtual channels should only be placed on a route if it is the *only* "best" route between two processors. If all "best" routes have zero tolerance, then one will be picked arbitrarily and *all* virtual channels will be routed on that one.
- A default `tolerance` value indicates that channels may be shared between the "best" routes between two processors.
- A `tolerance` value which is some multiple of the routing cost values in the network indicates that channels should be shared between the "best" routes and those routes with a higher cost but with tolerance values indicating that they are also acceptable.
- The maximum `tolerance` value indicates that *all* routes between two processors can be used for channels. This might lead to some very long routes being chosen.

Link quota

The third attribute – `linkquota` – controls how many links on a processor may be used to carry virtual channels to the processes on that processor. In the default case any of the links may be used. For each link which is used, a small additional memory overhead is incurred. On processors with very small amounts of memory it may be important to keep the memory overhead as low as possible.

The `linkquota` attribute can be set to a value in the range 0 to 4 inclusive. It should only be set to 0 if no virtual channels will be required by the processes on that processor. If it is set to 1, then the processes on the processor may use virtual channels, but it should be possible for the configurer to implement them all via *one* of the processor's links. Similarly for values of 2, 3, and 4 (although, obviously, setting the quota to 4 on a processor with four links has no effect).

The `linkquota` attribute is a *guide* to the configurer rather than an absolute directive. If a processor has a `linkquota` value of 1, but the processor provides the only route available for the implementation of a particular channel in the network, then the configurer will choose to route data through that processor, even though this will cause the link quota to be exceeded.

The `linkquota` is not intended as a method of avoiding routing through a processor; the `routecost` attribute should be used for that. Instead it is intended to indicate, on memory-critical processors, that the minimum overhead should be placed on them. The quota should reflect the requirements of the processes placed on that processor, and the routing costs in the network should be chosen so that other processors are used for through-routing. The link quotas will then be checked by the configurer as it sets up the multiplexing and routing processes. The configurer will output a warning message if it has exceeded a quota. The network can then be re-examined to see why this is happening.

The minimal spanning tree

There is one aspect of the implementation of virtual channels which may become evident when constraints are placed on how the configurer may route channels in the network. Normally the configurer can use any of the links in the network for virtual channels, so if the network is connected, then virtual channels can be routed from any processor to any other. However, (as described in section 9.2.1) it is possible to `place` a pair of opposing channels on a link in the network; in this case the link is used directly to implement those two channels, and cannot be used for virtual channels. Also the `routecost` attribute on selected processors in the network may prevent the use of some processors (and hence links) in the network for through-routing. If too many links are removed from the network in this way then it may become impossible to implement some of the virtual channels required.

So it is important to ensure that, for a set of processors in a network requiring virtual channels to be connected between them, there is a set of links connecting the processors over which virtual channels is allowed. This set of links will then be used by the configurer to construct a *minimal spanning tree* of links to ensure that it can always implement the virtual channels between these processors. Any additional links available for virtual channels will also be used to provide better routes between processors. If the configurer is unable to construct the route necessary to implement a requested virtual channel, it will give an error message.

A network may not require a single minimal spanning tree to cover the whole network; it depends on the virtual channel requirements of the configuration. For example, it might be possible to divide a configuration into two separate parts, each requiring virtual channels internally, but with a single pair of channels (which can be directly mapped onto a link) joining the two parts. In this case a minimal spanning tree of links is required for each of the two parts. These are known as *sub-networks*.

Summary of routing and placement attributes

The attributes are defined in more detail as follows:

- `routecost` - defines within the range `MIN_COST` = 1 to `MAX_COST` = 1000000 inclusive, the associated cost of routing virtual channels through a particular processor.

  If a value greater than `MAX_COST` (e.g. `INFINITE_COST` = 1000001) is specified then no through-routing will be permitted on that processor.

  If this attribute is not defined for a particular processor then the cost value `DEFAULT_COST` = 1000 will be assumed.

  `MAX_COST`, `MIN_COST`, `INFINITE_COST`, and `DEFAULT_COST` are predefined constants that can be used in any configuration source file for `icconf` (see appendix B).

- `tolerance` - controls with any value in the range `ZERO_TOLERANCE` = 0 to `MAX_TOLERANCE` = 1000000 inclusive, how much a particular processor can be used to provide load-sharing routing paths for other processors.

  The default value for this attribute is `DEFAULT_TOLERANCE` = 1. This allows the processor to implement alternate routes for through-routed channels with exactly the same total cost as the "best" route found between any two other processors.

  If the value `ZERO_TOLERANCE` is specified then the processor will only be used for through-routing if it lies on the "best" route found to implement virtual channels.

  If `tolerance` is set to `MAX_TOLERANCE` on all processors in the target network almost every possible route will be used to share the cost of carrying data between any pair of non-adjacent processors.

  `MAX_TOLERANCE`, `ZERO_TOLERANCE`, and `DEFAULT_TOLERANCE` are predefined constants that can be used in any configuration source file for `icconf` (see appendix B).

- `linkquota` - suggests the maximum number of links on the processor that should be used by the virtual channel routing system.

  `linkquota` can have the values 0 to 4 inclusive.

  A warning will be produced if the suggested `linkquota` for a node is exceeded. The `linkquota` will only be exceeded because of the requirements of through-routing data for other processors.

Prevention of through–routing via critical processors

If there are processors within the target network that are likely to be CPU-limited by the application, then it may be undesirable to allow virtual channels from surrounding processors to be routed through the performance-critical processors. In this case the `routecost` attribute for the critical processors should be set to `INFINITE_COST`. If this is done then no virtual channels can be through-routed via these processors.

Care must be taken to ensure that a minimal spanning tree of links is provided by the other processors in the network. If a particular processor should only be used for through-routing channels when absolutely necessary, then the `routecost` attribute on the processor can be set to some multiple of `DEFAULT_COST`. Alternatively the cost value can be explicitly set on the other processors. If for example, the multiple concerned is larger than the number of lower cost processors in the network then any route via those processors will be chosen in preference to a route via one of the high cost processors.

Use of additional processors for through–routing

There may be situations when the configurer is required to route all communications via a particular set of processors. For example:

- to emulate closely the communications structure that would be provided by dedicated hardware routing devices, or
- when a block of low performance processors is provided in the target network solely for the purposes of through-routing data for other processors.

This can be achieved in one of two ways;

- If the `routecost` of all processors, other than those intended as routers, is set to `INFINITE_COST` then the only processors that the configurer can use for through-routing are those left with the default routing cost. This technique has the advantage of guaranteeing that no through-routing will be done via the standard processors.
- If the `routecost` of all the routing processors are set to a small value e.g. `MIN_COST`, then any route via these processors will be used in preference to routes via processors with the default routing cost. This technique has the advantage that the normal processors can still be used by the configurer for routing channels that cannot be implemented by the nominated routing processors. Hence the nominated routing network need not provide full connectivity.

Generally the second method is preferred as it preserves the ability of the configurer of mapping an arbitrary application onto the target hardware.

Support for memory–critical systems

It may be desirable to ensure that for a particular processor the additional run-time overhead added by the configurer is kept to a minimum.

Normally the configurer spreads virtual channels running between a pair of processors across all routes that have equal cost. For each additional route employed additional support processes may be required and hence additional memory consumed on the target system.

This should not normally be a problem as the total cost of the maximal set of runtime processes that can be placed on the target system by the configurer consumes only a few thousand bytes more than the minimal set.

Some example figures of the minimum and maximum costs of both through-routing and multiplexing software on different word length transputers are shown below. (All sizes are in bytes):

Word Size	Function	Code	Min Stack	Max Stack
32 bits	Through-routing	716	768	2112
	Multiplexing	2104	784	2056
16 bits	Through-routing	724	512	1568
	Multiplexing	2118	524	1556

Multiplexing software is needed whenever a processor has virtual channels terminating on it. In the current system each opposing pair of virtual channels forming a virtual link will require approximately 120 bytes of local storage on a 32-bit processor and 80 bytes of storage on a 16-bit processor. **Note:** that extra overheads will be incurred if the `G` option is given to the configurer to allow interactive debugging of the application.

A particular case of the critical memory problem comes when the set of user processes on a particular processor do not in themselves require virtual channels at all, because the channels they use can be mapped directly onto the hardware links available. However, if the configurer decides to use through-routing then through-routing support processes will be added to the processor. In addition, to enable the available hardware links to be shared, some of the channels used on the processor may be implemented as virtual channels. In this case multiplexing software will also be required. In this special case the processor can be completely protected from run-time overheads by using the techniques described above for the '*Prevention of through–routing via critical processors*'.

A `linkquota` attribute can be specified on each processor in the target network. If the `linkquota` of a particular processor is specified as `1` and the `routecost` set to `INFINITE_COST`, then only a single hardware link will be used on the processor to provide all the virtual channels it uses. In addition the memory overheads of the virtual link system will be reduced to a minimum (minimal multiplexer only).

If `linkquota` is set to 1 on all processors in the target system then the minimal spanning tree of links will be used to support all virtual channels required. Warnings will be produced in this case for all processors that have had more than `linkquota` links used on them; this is because all processors cannot be chosen as "leaves" in the spanning tree.

If both performance and memory size are a problem in a particular system it is likely that the user will have to tune the **linkquota** and **tolerance** parameters of many processors in order to get the best result.

9.2.4 Example – optimized filter test program

Figure 9.3 describes an example configuration that needs to be placed onto a network of six processors (figure 9.4). The function of the program is to test the two filter components which are limited by the speed of the processors concerned. The source for the example is supplied in the **examples/router** subdirectory

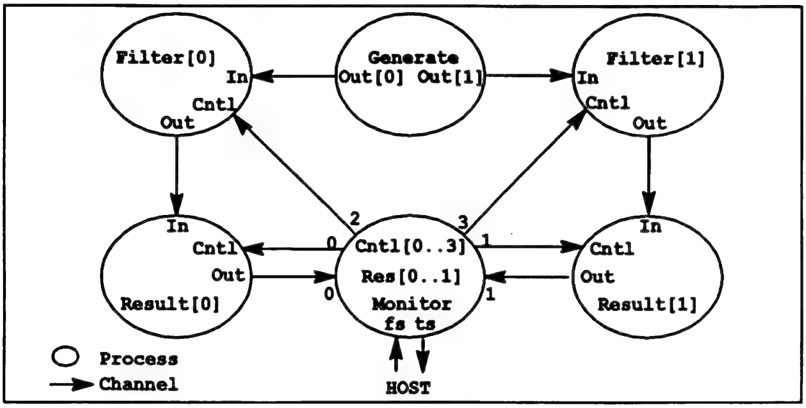


Figure 9.3 Example filter test program

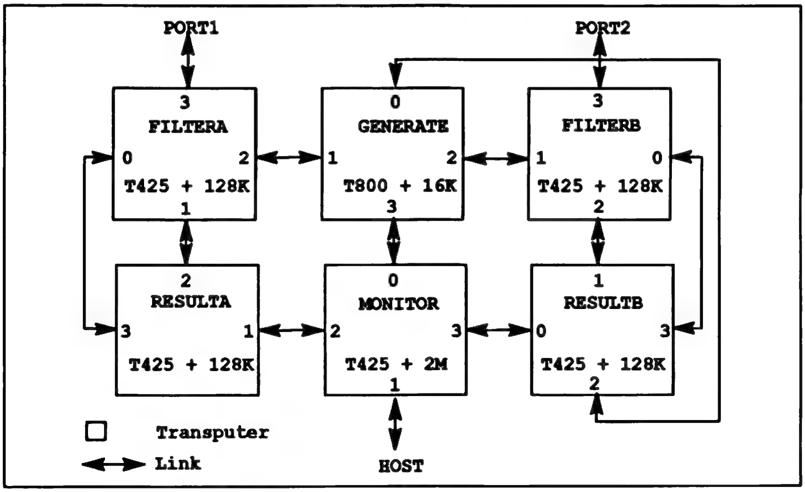


Figure 9.4 Example filter test hardware

This is not a real program but has been constructed to demonstrate many of the features for optimization described in the previous sections, within a comparatively small and simple system. The basic configuration description is as follows:

```
/* Hardware description for specialised sub-system */

T800 (memory = 32K)  GENERATE;
T425 (memory = 128K) FILTERA;
T425 (memory = 128K) FILTERB;
T425 (memory = 128K) RESULTA;
T425 (memory = 128K) RESULTB;
T425 (memory = 2M)   MONITOR;

edge port1, port2;

connect host             to MONITOR.link[1];
connect MONITOR.link[2]  to RESULTA.link[1];
connect RESULTA.link[2]  to FILTERA.link[1];
connect FILTERA.link[2]  to GENERATE.link[1];
connect GENERATE.link[2] to FILTERB.link[1];
connect FILTERB.link[2]  to RESULTB.link[1];
connect RESULTB.link[0]  to MONITOR.link[3];

connect RESULTA.link[3]  to FILTERA.link[0];
connect RESULTB.link[3]  to FILTERB.link[0];

connect MONITOR.link[0]  to GENERATE.link[3];

connect GENERATE.link[0] to RESULTB.link[2];
connect FILTERA.link[3]  to port1;
connect FILTERB.link[3]  to port2;

/* Software description for filter test program */

input fs;
output ts;

process (stacksize = 2k, heapsize = 16k,
         interface(input fs, output ts, input Res[2],
                   output Cntl[4])) Monitor;

process (stacksize = 2k, heapsize = 16k,
         interface(input In, output Out,
                   input Cntl)) Result[2];

process (stacksize = 2k, heapsize = 16k,
         interface(input In, output Out,
                   input Cntl)) Filter[2];

process (stacksize = 1k, heapsize = 4k,
         interface(output Out[2])) Generate;

rep i = 0 for 2
{
  connect Monitor.Cntl[i]    to Result[i].Cntl;
  connect Monitor.Cntl[i+2]  to Filter[i].Cntl;

  connect Result[i].Out      to Monitor.Res[i];

  connect Filter[i].Out      to Result[i].In;

  connect Generate.Out[i]    to Filter[i].In;
}

connect Monitor.fs to fs;
connect Monitor.ts to ts;

/* Mapping description */

place Generate   on GENERATE;
place Filter[0]  on FILTERA;
place Filter[1]  on FILTERB;
place Result[0]  on RESULTA;
place Result[1]  on RESULTB;
place Monitor    on MONITOR;

use "generate.lku" for Generate;
use "filter.lku"   for Filter[0];
use "filter.lku"   for Filter[1];
use "result.lku"   for Result[0];
use "result.lku"   for Result[1];
use "monitor.lku"  for Monitor;

place fs on host;
place ts on host;
```

However, for this real-time program to actually work correctly a number of optimization features of the configurer have to be exploited to ensure the right routing decisions are made:

- GENERATE has no memory space available to carry the overheads of routing software and requires no virtual channels itself, so setting `routecost` to `INFINITE_COST` prevents routing software being placed on it.
- FILTERA and FILTERB must be operated in a state as close as possible to the real case, where all their channels are placed onto hardware links. The main data path through the `Filter` component must operate at hardware data rates, so the `In` and `Out` channels must both be placed onto hardware links to guarantee the required performance. The `Cntl` channel

which carries a small amount of parameterization data can, however, be implemented as a virtual channel without significant effect.

Hence the following text is added to the basic configuration source:

```
/* Mapping optimisation */

/* Prevent through routing via GENERATE */
GENERATE (routecost = INFINITE_COST);

/* Ensure minimum overhead on FILTERA */
FILTERA (routecost = INFINITE_COST, linkquota = 1);

/* Ensure minimum overhead on FILTERB */
FILTERB (routecost = INFINITE_COST, linkquota = 1);

/* Optimise Generate to Filter 0 Path */
place Generate.Out[0] on GENERATE.link[1];
place Filter[0].In    on FILTERA.link[2];

/* Optimise Generate to Filter 1 Path */
place Generate.Out[1] on GENERATE.link[2];
place Filter[1].In    on FILTERB.link[1];

/* Optimise Filter to Result 0 Path */
place Filter[0].Out on FILTERA.link[1];
place Result[0].In  on RESULTA.link[2];

/* Optimise Filter to Result 1 Path */
place Filter[1].Out on FILTERB.link[2];
place Result[1].In  on RESULTB.link[1];

/* Use otherwise unspecified linkquotas to check
   overheads on GENERATE, RESULTA, and RESULTB */
GENERATE (linkquota = 0);
RESULTA (linkquota = 2);
RESULTB (linkquota = 2);
```

10 Mixed language programming

This chapter describes the mechanisms for mixing code modules written in different high level languages. It is divided into two parts. The first part discusses how to call procedures and functions written in one language from another language. This includes details of the library procedures provided to allow occam programs to call C functions which require use of static or heap memory.

The second part describes how complete C programs can be called as if they were occam processes with a standard channel interface.

10.1 Mixed language programs

For many applications it is appropriate to write the software using more than one programming language. For example, a particular algorithm may be better expressed in a specific language, or application modules may already exist in particular languages. In either case a well defined mechanism for mixing languages within a single system is desirable.

The toolset provides a clean and simple basis for mixing languages on transputer networks. Independent software processes can be written in different languages, compiled and linked using a common set of tools, and the linked modules placed anywhere on a network of transputers using a configuration description. Compiler pragmas are provided to allow code to be imported with the correct calling conventions, and to translate names so they are valid in the calling language.

Code written in other languages can be used as external routines in a program, providing the language calling conventions are honored, and no conflicts of name occur.

There are a number of issues to be considered when mixing languages. These are:

- The declaration of the external routine — in order for the calling program to be able to correctly call an external routine, it must have a description of the interface to the routine. The way in which this is done depends on the language being used.

- The translation of names — programming languages differ in the legal character set for identifiers and symbolic names. Thus, names acceptable in one language may not be valid in another. To avoid these problems compiler pragmas are provided to perform name translations.

- The calling conventions of the languages — including passing the address of the static area and the types of the parameters in the two languages.
- The types returned by functions.
- The presence, or otherwise, of a *static area* in each language (this is discussed in more detail below).
- The libraries to be used when linking the complete program.

These issues are discussed in more detail in the the following sections.

Note: When mixing languages, the external procedures must not do any host communications. All i/o should be performed by the calling program. The external procedures *can* however perform channel communications with other processes.

10.1.1 Declaring external routines

In order to properly call a separately compiled procedure or function, the compiler needs to be given information about the external routine. In C this is done by declaring the function as external, for example:

```
extern int f (int a, int b);
extern void p1 (char c);
```

The functions should be declared as prototypes, including the types of parameters, to ensure that the actual parameters are converted to the specified types. If the functions are declared without the parameter types then the default C argument type promotions will take place.

The occam compiler uses a pragma to provide information about external procedures and functions. The syntax of this is:

`#PRAGMA EXTERNAL` "*formal declaration* = *workspace* [, *vectorspace*]"

The optional parameter *vectorspace* is not required for C functions.

For example:

```
#PRAGMA EXTERNAL "PROC p1 (VAL BYTE c) = 20"
#PRAGMA EXTERNAL "PROC p2 (BYTE x, y) = 40, 100"
#PRAGMA EXTERNAL "INT FUNCTION f (VAL INT a, b) = 50"
```

A void function in C is equivalent to a procedure in occam.

10.1.2 Translating identifiers

Because the syntax of valid identifiers can vary from one language to another, compiler pragmas are provided in C and occam to allow the names used in a source file to differ from those used externally.

The pragma can be used to change the name which is used in the object code to reference an external routine. For example, a C program which needs to call an occam function called '`get.next`' could use the following to convert the name into a valid C identifier:

```
#pragma IMS_translate(get_next, "get.next")

extern void get_next(int *n, Channel *in);
```

Alternatively the pragma could be used to change the name 'exported' from the occam code:

```
#PRAGMA TRANSLATE(get.next, "get_next")

PROC get.next (INT next, CHAN input)
  ⋮
:
```

In this case, the object file will contain the name '`get_next`' and the procedure can only be called by this name.

10.1.3 Parameter passing

The two issues in passing parameters between languages are, firstly, the types of the formal and actual parameters (including whether they are passed by *value* or by *reference*) and, secondly, the use of a static area by each language. These are described in more detail below.

Parameter compatibility

Correct parameter passing depends on the compatibility of data types between languages. See the language implementation chapters of the appropriate *Language and libraries* manuals for details of the implementation of types and how parameters are passed.

The way in which parameters are passed — either as a copy of the data (by *value*) or a pointer to the data (by *reference*) — involves two issues: the semantics of the language, and the actual implementation.

C: All parameters are passed by value. Arrays are passed as pointers to the base type of the array. It is possible to pass pointers to variables which gives the effect of passing by reference.

occam: parameters are either VAL parameters or non–VAL parameters. VAL parameters may be implemented by passing by value, or by passing a pointer. The latter will happen when the size of the parameter is larger than the word length of the processor and will therefore depend on the data type and the processor type.

Types can be considered to be compatible if they have the same interpretation, are the same size and are passed in the same way. For example, a C parameter of type `int` is compatible with an occam `VAL INT` parameter. Similarly, as an occam `INT` parameter is passed as a pointer it is compatible with a C `int *` parameter.

Partial lists of type compatibilities are shown in tables 10.1 and 10.2.

occam type	C type

`VAL BYTE`	`char`
	`unsigned char`

`BYTE`	`char *`
	`unsigned char *`

`VAL INT16`	`short int`
`INT16`	`short int *`
`VAL INT`	`int`
`INT`	`int *`
`INT32`	`long int *`
`REAL32`	`float *`

`VAL REAL64`
`REAL64`
	`double *`

`CHAN`	`Channel *`
`TIMER`	No parameter required

Table 10.1 Type equivalents for all processors

occam type	C type	
	16 bit processor	32 bit processor
`VAL INT32`	`long int *`	`long int`
`VAL REAL32`	`float *`	`float`

Table 10.2 Type equivalents dependent on processor word length

Range checking

It is important to ensure that parameters passed to occam procedures and functions have values within the legal range for the type. For example, when passing to a formal parameter of type `BYTE` the value must be in the range 0 through 255.

Violation of this rule is liable to cause a runtime range check error in the occam code.

occam timers

An occam `TIMER` parameter should have *no* associated actual parameter. For example, the procedure `PROC p (VAL INT p1, TIMER t, VAL INT p2)` could be called from C simply as: `p (p1, p2)` (assuming the `nolink` pragma has been used — see below).

10.1.4 Global static base parameter

C uses an area of memory for static data. This requires a parameter to be passed to the called function to enable it to access the static area — this parameter is known as the Global Static Base or GSB. This parameter is added automatically by the compiler and is not normally visible to the programmer.

occam differs from C in that it does not use a static or heap area and so does not expect a GSB parameter to be passed to procedures. Similarly, occam programs do not pass a GSB pointer when procedures are called. In order to allow calls to work correctly between languages the presence of the GSB parameter must be taken into account.

There are two possible solutions to this problem:

1. A dummy GSB parameter can be provided in occam.
2. A compiler pragma can be used in the C program to specify that a function does not require a GSB parameter.
3. When calling occam from C, make use of the `call_without_gsb` function (see chapter 2 of the C *Language and libraries* manual).

The first two techniques can be used either on the routine being called or in the calling program, whichever is more appropriate.

In the examples below which show C functions called from occam, it is assumed that the C code does not use any static or heap memory. However, it will often be necessary for the occam calling program to allocate some memory for use by the C code as the static or heap area; a pointer to this memory is then passed as the first parameter when the function is called. This technique is described in more detail in section 10.1.7 below.

Method 1 — dummy GSB parameter.

A dummy parameter can be used either as a formal parameter for procedures which are to be called from C, or as an actual parameter for C functions which are being called from occam. For example the following occam function can be directly called from a C program:

```
INT FUNCTION ocfunc(VAL INT GSB, arg1, arg2)
  -- Note: dummy parameter GSB is not used
  INT return:
  VALOF
    :
    RESULT return
:
```

Note: because the dummy parameter is not used, the occam compiler will generate a warning message but correct object code is still generated.

To call this version of `ocfunc` from a C program it is declared as an `extern` function (without the GSB parameter) and then called normally:

```
/* declare function as external */
extern int ocfunc(int arg1, int arg2);
⋮
/* call function */
ret = ocfunc(x, y);
```

The same method can be used to call a C function from occam by passing a dummy first parameter of type `INT`. For example the C function:

```
void cfun(int a)
{
⋮
}
```

Could be called from occam in the following way:

```
#PRAGMA EXTERNAL "PROC cfun (VAL INT GSB, x) = 20"
⋮
  VAL INT GSB IS 0:
  cfun(GSB, 42)
```

Method 2 — `nolink` pragma

In order to simplify mixing occam and C, the INMOS C compiler provides the `IMS_nolink` pragma which directs the specified function to be compiled without the static link parameter. Any *calls* of the function, within the scope of the pragma, will not have the GSB added to the parameter list. If the function is *defined* within the scope of the pragma then it will be compiled without the requirement for a static link parameter (the compiler will flag a serious error if the function requires access to static data).

As an example, consider the occam function `ocfunc` below:

```
INT FUNCTION ocfunc(VAL INT arg1, arg2)
  INT ret :
  VALOF
     ⋮
    RESULT ret
:
```

To call `ocfunc` from a C program it must first be declared as an `extern` function and then specified as not requiring the GSB parameter:

```
/* declare function as external */
extern int ocfunc(int arg1, int arg2);

/* specify that function has no GSB parameter */
#pragma IMS_nolink(ocfunc)

/* call function */
ret = ocfunc(x, y);
```

The same technique can be used to compile a C function which does not require a GSB parameter so that it can be called directly from occam. As an example, consider the C function below:

```
/* declare function before referencing */
void cfun(int a);

/* specify that function has no GSB parameter */
#pragma IMS_nolink(cfun)

/* define the function */
void cfun(int a)
{
:
}
```

This can be called from occam in the following way:

```
#PRAGMA EXTERNAL "PROC cfun (VAL INT x) = 20"
:
  cfun(42)
```

Method 3 — using `call_without_gsb` function

This method is applicable only when dynamically loading code using the ANSI C Toolset. It is described in chapter 12 and an example is given in section 12.6 of the *ANSI C Toolset User Guide*.

10.1.5 Function return values

When functions are being called it is also necessary for the return types to be compatible.

The definition of compatibility for function return types is stricter than that for parameters. It is possible (though probably not sensible) to pass any type of the correct size as a parameter. However, floating point and integer function results are returned in different ways (depending on the processor type) and so it is essential to ensure that the types of function return values are strictly equivalent. A partial list of equivalents is given in table 10.3 for guidance.

occam function type	C function type

`BYTE`	`char`
	`unsigned char`

`INT32`	`long int`
`INT`	`int`
`REAL32`	`float`
`REAL64`	`double`

Table 10.3 Function return types

Note: a C function of type `void` must be called from occam as a procedure. Similarly an occam procedure must be called from C as a void function.

Restrictions on functions that may be called

Because occam functions can only have `VAL` parameters, and these do not always have C equivalents, there are restrictions on the types of occam functions that can be called from C and vice-versa. For example, there are no equivalents of the occam `BOOL` type and so functions which require this type of parameter cannot easily be called.

Similarly, because C functions can only return a single value, only occam functions with a single return value can be called from C.

occam cannot call C functions which return structure types.

10.1.6 Linking the program

After all the component parts of the program have been compiled, they must be linked together with any libraries required. The libraries that are required will depend on a number of factors such as the language that the main (calling) program is written in, whether the program communicates with the host, which library routines are used by the different language modules. Some guidelines for various configurations are given below.

Calling occam from C

When calling occam code from a C program, then the following library files must be linked with the compiled occam and C code.

- The C runtime library

  If the program uses the host file server then the *full* runtime library must be used. This can be linked in by using the linker indirect file `cstartup.lnk`.

  If the program does not use the host file server then the *reduced* runtime library must be used. This can be linked in by using the linker indirect file `cstartrd.lnk`.

- The standard occam compiler libraries will be required by most occam code. These libraries can be linked in by using the appropriate `occamx.lnk` linker indirect file.

- Any other C or occam modules or libraries referenced by the program must also be linked in.

Calling C from occam

When calling C code from an occam program, then the following library files must be linked with the compiled C and occam code.

- The standard occam compiler libraries can be linked in by using the appropriate `occamx.lnk` linker indirect file.

- If the main program is written in occam and allocates static or heap memory for C functions using the library procedures described in section 10.1.7, then the library `callc.lib` must be linked in.

- Any other C or occam libraries used must also be linked in.

- The *reduced* C library must be used as the called functions cannot make any host file server requests. The reduced runtime library can be linked in by using the `clibsrd.lnk` linker control file.

10.1.7 Allocating memory for C functions called from other languages

The C runtime environment automatically provides C programs with a static area (for holding static data and external variables) and a heap area (for memory allocation). However occam does not provide these and so this memory must be explicitly allocated by the calling program before C functions are called. Four routines in the occam library `callc.lib` are used to set up and terminate C static and heap areas from occam for C functions that require them.

The static area

C static data is stored in a reserved area of memory called the static area which must be set up by the system and initialized. Each C function which uses static data needs to be able to find this area. In order to do this, every C function is passed, as the first parameter, a pointer to the start of the static area, the global static base (GSB). The static area must be set up and the GSB parameter passed explicitly by the calling occam code. This means that a call to a C function from occam will have one extra parameter compared to an equivalent call from C.

The heap area

The heap area is that area of memory from which the C memory allocation functions reserve their memory space. It is separate from the static area and requires a static area to be previously allocated because information about the heap is held in static variables.

The heap need not be set up if it is not required, but remember that it may be used implicitly by a library call.

Providing static and heap

Some simple C functions may not require static or heap areas and may be called more easily without using the special library routines. When calling a C function therefore, the first step is to decide whether static and heap areas are required.

Deciding whether a static area is required

For many C functions it may not be immediately obvious whether static or heap is required (the heap area requires a previously set-up static area). For example, some, but not all, library functions require static and heap areas and so, because it would be difficult to distinguish those that do, a static and heap area should be assumed whenever a library function is called.

Because of the difficulty in covering all types of functions, the following series of rules is offered as a way of determining whether a function requires static or heap. The rules include the most common reasons for a C function requiring static or heap memory.

- If the function uses static variables then static is required.
- If the function accesses external variables then static is required.
- If the function includes an automatic structure or union initializer then static is required.
- If the function uses any functions from the runtime library then static and heap may be required.

Functions which fail all the above tests will probably not require static or heap, and can be called without using any of the static or heap library functions.

Calling functions which do not require static or heap

C functions which do not require static or heap can be called as described in section 10.1.4.

Calling functions which do require static or heap

For C functions which require static and/or heap the space must be set up in the occam code before the function is called, and terminated when no longer required. These operations are performed by procedures supplied in the library `callc.lib`. This library is supplied as part of the ANSI C toolset — do *not* use any previous version of the library which was supplied as part of an occam toolset.

The library `callc.lib` provides four occam procedures for initializing static and heap areas and terminating them after use. The routines are summarized in table 10.4 and described in more detail below.

Procedure	Description
`init.static`	Initializes an area of memory for use as the static area.
`init.heap`	Initializes an area for use as the heap area.
`terminate.heap.use`	Terminates heap usage.
`terminate.static.use`	Terminates static usage.

Table 10.4 Library procedures to support memory allocation

```
PROC init.static([]INT static.area, INT required.size, GSB)
```

`init.static` is used to set aside and initialize an area of memory for use as a C static area before any C functions are called. The static area is declared as an integer array in the calling occam program.

Two integer values are returned in the procedure parameters:

`required.size`	The number of words of static space required.
`GSB`	A pointer to the base of the array which will act as the global static base.

Note: the size of the integer array is equivalent to the number of words of static space required. One element of the integer array is equivalent to one word of memory. If an error occurs on initializing the static area the value `MOSTPOS INT` is returned instead of the required size.

The procedure can be used to check the size of static area required by checking the value returned in the second parameter. For example:

```
#USE "callc.lib"

INT required.size, GSB:
[STATIC.SIZE]INT static.area:

SEQ
  init.static(static.area, required.size, GSB)
  IF
    required.size > STATIC.SIZE
      ...  not enough space reserved
    TRUE
      ...  array is big enough
```

Another possible way of using `init.static` is to reserve a large amount of memory for use by the C function. To do this an initial call to `init.static` would be made with an array size of zero to obtain the required size, followed by a second call which would set up a segment of memory as the static area. The rest of the memory could be used by the occam program for its own purposes, perhaps to allocate the C heap. For example:

```
#USE "callc.lib"

INT required, GSB:
[VERY.BIG.NUMBER]INT memory :

SEQ
  -- check the static requirement
  init.static([memory FROM 0 FOR 0], required, GSB)

  -- allocate required amount of memory for static
  static.area IS [memory FROM 0 FOR required]:
  -- rest is available for other purposes
  memory.left IS [memory FROM required FOR
                  (VERY.BIG.NUMBER - required)]:
  SEQ
    -- now use allocated memory as static
    init.static(static.area, required, GSB)
    ...  rest of program
```

```
PROC init.heap(VAL INT GSB, []INT heap.area)
```

`init.heap` is used to set aside an area of memory for use as a C heap before any C functions are called. The first argument is the GSB pointer returned by `init.static`, which is required because the memory allocation routines make use of static data.

Like the static area, the heap area is declared as an integer array. This array must be large enough to accommodate all calls to the C memory allocation functions.

The size of the integer array is equivalent to the number of words of heap area required. One element of the integer array is equivalent to one word of memory.

If the heap is used by a function before `init.heap` has been called the C memory allocation functions will fail with their normal error returns.

```
PROC terminate.heap.use(VAL INT GSB)
```

`terminate.heap.use` should be called when the heap is no longer required, i.e. when no more C functions will be called. It provides a clean way of terminating the use of the heap.

Once the heap terminate procedure has been called, the state of the heap is undefined.

`terminate.heap.use` must be called *before* terminating the static area because the heap is accessed using static variables.

```
PROC terminate.static.use(VAL INT GSB)
```

`terminate.static.use` should be called when the static area is no longer required, i.e. when no further calls to C will be made. It provides a clean way of ending the use of the C static area.

Once the static terminate procedure has been called, the state of the static area is undefined.

Example

The following example illustrates how these library procedures can be used to set up and terminate the static and heap areas for a C function. The C function to be called is:

```
#include <stdlib.h>

int c_func(int n, int release){

  static int *ptr = NULL;
  int i;

  if (ptr == NULL){
    ptr = (int *) malloc(n);

    if (ptr == NULL)
      return 1;
  }

  for (i = 0; i < n / sizeof(int); i++)
      ptr[i] = i;

  if (release){
    free (ptr);
    ptr = NULL;
  }

  return 0;
}
```

The occam code to call this function (on a 32 bit transputer) is shown below:

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"
#USE "callc.lib"  -- the 'calling C' functions.

#PRAGMA TRANSLATE C "c_func"

-- declare the C function as an occam descriptor.

#PRAGMA EXTERNAL "INT FUNCTION C(VAL INT GSB,x,free) = 200"

PROC mixed (CHAN OF SP fs, ts, []INT freemem)

  INT GSB, required.size :

  -- Allow very large static and heap area sizes
  VAL static.size IS 4000 :
  VAL heap.size   IS 4000 :
  [static.size]INT static.area :
  [heap.size]INT   heap.area :

  SEQ
    -- set up static.area as the static area
    init.static(static.area, required.size, GSB)
    -- now check for error
    IF
      required.size > static.size
        so.write.string(fs, ts,
                 "error initialising static*n")
      TRUE
        INT fail:
        SEQ
          -- Set up the heap area.
          -- Note that GSB is the first parameter
          init.heap(GSB, heap.area)

          -- Call the C function. Note that the GSB
          -- is passed as the first parameter.
          fail := C (GSB, 20000, 0)
          IF
            fail = 0
              so.write.string(fs, ts, "malloc OK*n")
            TRUE
              so.write.string(fs, ts, "malloc failed*n")
    -- now tidy up the stack and heap allocated
    terminate.heap.use(GSB)
    terminate.static.use(GSB)
    -- and exit
    so.exit(fs, ts, sps.success)
:
```

The occam program must be compiled and then linked with the compiled C function, the memory allocation library, the reduced C runtime library, the occam host i/o library, and the standard occam libraries. In this example (assuming that the C source code is in a file called `cfunc.c` and the occam source is in a file called `mixed.occ`) the set of files to be linked is:

`mixed.tco`	compiled occam program
`cfunc.tco`	compiled C function
`clibsrd.lnk`	linker indirect file for the C reduced runtime library
`hostio.lib`	occam i/o library
`callc.lib`	call C library
`occama.lnk`	linker indirect file listing standard occam libraries for code compiled for transputer class TA

The linker allows files to either be specified on the command line or listed in an indirect file. Because there are several files required in this instance, it may be easier to supply a linker indirect file. This file can also include a `#mainentry` directive to define the entry point of the program, in this case the top level occam procedure "`mixed`". To do this create a text file called `callc.lnk`, containing the following lines:

```
mixed.tco
cfunc.tco
#include clibsrd.lnk
hostio.lib
callc.lib
#include occama.lnk
#mainentry mixed
```

The correct linker command line (using the default processor T414 in HALT mode) would be as follows:

```
ilink -f callc.lnk                                    (UNIX)

ilink /f callc.lnk                                    (MS-DOS/VMS)
```

Details of the operation of the linker can be found in chapter 9 in the *Toolset Reference Manual.*

10.1.8 Restrictions and caveats

General

A number of restrictions must be observed when calling routines written in one language from a program in a different language:

1 The formal and actual parameters (and function return types) must be compatible. See sections 10.1.3 and 10.1.5 for more detail.

2 As occam does not have 'external' variables, there can be no common data between the calling program and the called routine. Therefore, the only way that data can be transferred between them is by means of parameters (and return values). The called procedure may also use channels to communicate with other parts of the program that are running in parallel.

3 No function or procedure which requires direct communication with the *host file server* may be called.

Rules for importing C code

The following restrictions apply to C functions which are to be called from an occam program:

1 Stack checking should not be enabled in any C function to be called from occam.

2 Only C functions linked with the reduced C runtime library, can be called from occam, i.e. those which do not require any server communication.

3 The following functions cannot be called in the imported C code:

```
clock()

exit()

exit_terminate()

exit_noterminate()

exit_repeat()

get_detail_of_free_stack_space()
```

Rules for importing occam code

There are certain rules which govern the calling of occam code from C:

1 occam functions that return more than a single value may *not* be called.

2 The occam procedure or function to be called must be at the *outer level* of a compiled module.

3 `INLINE` procedures and functions cannot be called from C.

4 The occam code must not use vector space, or call any other occam code which uses vector space. Arrays, if used, should be explicitly placed within workspace or the code should be compiled with the `V` option to disable the use of separate vector space.

   Some occam libraries supplied with the occam 2 toolset use vector space and therefore cannot be called from C. These are:

```
hostio.lib  streamio.lib
msdos.lib   streamco.lib
```

5 There must be enough workspace for the called procedures or functions on the stack of the calling program. It is the programmer's responsibility to ensure that this is the case.

6 There must be no *aliasing* between the parameters to occam functions or procedures and the destination of the result. In other words the same variable must not be used as both a parameter which will be read, and as a result. The occam compiler checks that this is so for occam procedures and functions called within an occam program.

   The presence or absence of alias checking when the occam code is compiled has no effect on this rule.

   As an example consider the occam function:

```
INT FUNCTION succ (VAL INT n) IS n + 1 :
```

   If this is called from within an occam program, the compiler will check to see whether the parameter and result are aliased; if they are then the compiler will generate temporary variables as necessary. So, for example, the occam call `i := succ(i)` may be compiled with a temporary variable for the function result, which is then copied to the variable `i`. The C compiler is not able to perform these checks and so, if this function is called from C, it is up to the programmer to ensure that there is no aliasing. A suitable calling sequence could be:

```
int tmp;

tmp = succ(i);
i = tmp;
```

   Note that there may be mutual aliasing between `VAL` parameters as these are only read, not written.

10.2 occam interface procedures

The following sections describe a set of interfaces provided to allow complete programs written in C to be called from occam. This might be done for various reasons, for example to allow a C program to be used with the occam configurer `occonf`, or to provide some simple modification of the runtime environment of the program — e.g. initializing some external hardware before the application code starts, or intercepting the program's communications with the host file server.

By specifying the appropriate entry point for a C program, it is given an occam-like procedural interface allowing the program to be called from an occam program. The code produced in this way is known as an *occam equivalent process* as it makes the program look like an occam process with channels for input and output.

10.2.1 Interface code

The occam interface code described here provides a number of fixed interfaces to a C program. There are three types of interface code, known as types 1, 2, and 3. Descriptions and process diagrams for the three interfaces are given below.

Type 1

This interface is used when the C program runs on a single transputer and communicates only with the host file server. This interface is used with the full version of the C runtime library.

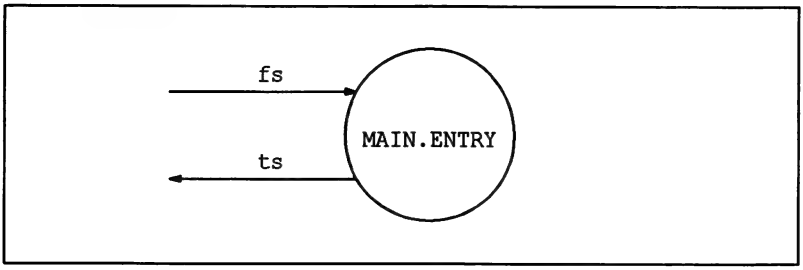


Figure 10.1 Type 1 interface

Type 2

This interface is used when the C program communicates with other processes as well as the host file server. This interface is used with the full version of the C runtime library.

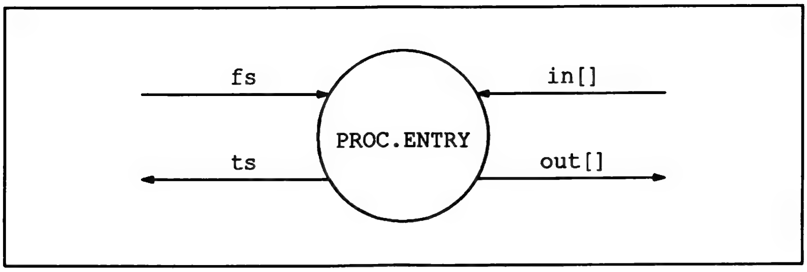


Figure 10.2 Type 2 interface

Type 3

This interface is similar to the type 2 interface except that there is no access to the host file server. The interface must be used with the *reduced* version of the runtime library, which does not contain any functions which require access to **iserver** facilities such as the host file system.

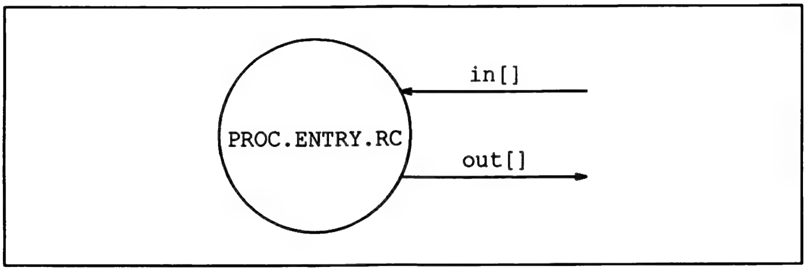


Figure 10.3 Type 3 interface

Channel arrays

The Type 2 and type 3 interfaces have arrays of channels which enable the C program to communicate with other processes in the program. These arrays are mapped directly onto the channel arrays which form part of the standard parameter list of the C **main** function (see section 10.2.7).

These channel arrays actually appear as arrays of integers in the occam parameter lists — this allows *pointers* to channels to be passed to the C program which provides a more flexible way of mapping channels onto the arrays. Because occam does not support pointers directly, two library procedures are provided to assign channel pointers to array elements (for more information on these, see the examples below and the *occam language and libraries* manual).

Reserved channels

Two of the input channels and two of the output channels in the Type 2 and Type 3 occam interface procedures (i.e. `in[0]`, `in[1]`, `out[0]` and `out[1]`) are reserved. No program should use these channels. They are reserved as follows:

`out[0]`	Reserved for diagnostic output.
`in[0]`	Reserved for diagnostic input.
`out[1]`	Messages from the runtime library to the host file server.
`in[1]`	Responses from the host file server to the runtime library.

10.2.2 Parameters to the C program

Parameters to the C `main` function are described by the following function prototype:

```
#include <channel.h>

int main (int argc, char *argv[], char *envp[],
          Channel *in[], int inlen,
          Channel *out[], int outlen);
```

Where:

- `argc` — the number of arguments passed to the program from the command line, including the program name.
- `argv` — an array of pointers to those arguments.

  Note: for programs linked with the reduced runtime library (i.e. using the Type 3 interface), `argc` is set to 1 and the first element of `argv` is a pointer to an empty string.
- `envp` — included for compatibility with previous toolsets — in this implementation, this parameter is always set to `NULL`.
- `in` — an array of input channels.
- `inlen` — the size of the array `in`.
- `out` — an array of output channels.
- `outlen` — the size of the array `out`.

The channel arrays `in` and `out` in the C program are passed from the interface procedures, and can be set up as described below. Where applicable, these channels can be used by the C code to communicate via channels passed in from the calling occam program. Note, however, that the first two elements in the arrays are reserved for use by the C program's runtime system and cannot be used by the application program.

10.2.3 Stack and heap requirements

Data storage (workspace) requirements for C programs are provided by arrays in the occam code. Stack, static and heap requirements vary from program to program. The workspace arrays passed to the program must be large enough to accommodate:

- the stack needed by the program when it runs
- all the static data required by the program
- the heap used by the program and the runtime libraries.

Stack overflow may lead to unpredictable behavior by the program. For this reason it is best to run a program initially with a large combined stack and heap. Later, after the program has been run to determine stack and heap usage, it can be modified to use a separate stack and heap of the appropriate sizes. The use of a separate array for the stack allows the stack to be placed in the transputer's internal memory to optimize the performance of the program. Methods for optimizing memory usage are described in two INMOS technical notes: number 17 *Performance maximization*, and number 55 *Using the occam toolsets with non-occam applications*. A minimum stack size of 512 words is recommended.

Stack overflow detection

Failure or unpredictable behavior of programs may be due to stack overflow. To obtain an estimate of the amount of stack used by a program:

1. Build all C code with stack checking enabled.
2. Call the function `max_stack_usage` at the end of the program, this will return an approximation of the amount of stack used by the program.

A test for stack overflow in a program is to use the procedure outlined below:

1. Initialize the bottom few words of the stack (a falling stack is used) to some easily recognizable pattern of values.
2. Run the program and, after it crashes, use the debugger to examine the values in the stack. If the values you initialized have been changed then stack overflow is likely.
3. Increase the stack size and try again.

A similar method can be used to determine static data and heap requirements, except that these are allocated upwards in memory. The following occam fragment gives an example of initializing the bottom of the stack:

```
SEQ i = 0 FOR SIZE ws1
  ws1[i] := #DEFACED
```

Stack overflow in the C parts of the program can also be detected by using the stack checking mechanism built into the C compiler and libraries.

10.2.4 Type 1 interface definition

The Type 1 interface is used when the C program does not communicate with any other process apart from the host file server.

The parameters for the Type 1 procedure are: a pair of channels to communicate with the host file server; and two arrays to provide the C program's heap, static and stack space.

Procedural interface

The Type 1 occam interface is defined as follows:

```
PROC MAIN.ENTRY (CHAN OF SP fs, ts,
                 []INT free.memory,
                 []INT stack.memory)
```

The parameters to this procedure are:

- `fs` — a channel from the host file server to the C program.
- `ts` — a channel from the C program to the host file server.

  The channels `fs` and `ts` are connected to the channels `in[1]` and `out[1]` which are passed as parameters to the C program — these are provided for the use of the C runtime libraries only, and should not be used by the application code.
- `free.memory` — used by the C program for its heap and static areas.

  This array is generally used to pass the free memory which is available to the C program after the all the code has been loaded.
- `stack.memory` — used by the C program for its runtime stack (if the size of the array is non-zero).

  If the size of the `stack.memory` array is zero then the `free.memory` array is used for the program's runtime stack as well as for the static and heap data areas.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to 2
- `in[0]` and `out[0]` are set to NULL

- `in[1]` is a pointer to the `fs` channel and is used by the C runtime system to communicate with the host
- `out[1]` is a pointer to the `ts` channel and is used by the C runtime system to communicate with the host

Example

The following example is an occam procedure, `call.prog1`, which calls a C program via the `MAIN.ENTRY` procedure interface:

```
#INCLUDE "hostio.inc"

PROC call.prog1 (CHAN OF SP fs, ts)

  #USE "centry.lib"           -- C interface code

  [100000]INT heap :          -- static and heap space
  [1024]INT   stack :         -- stack for program
  PLACE stack IN WORKSPACE :  -- Put on chip

  -- call program
  MAIN.ENTRY(fs, ts, heap, stack)
:
```

10.2.5 Type 2 interface definition

The Type 2 interface is used when building a program that will communicate with other processes as well as with the host file server.

The parameters for the Type 2 procedure are: a pair of channels to communicate with the host file server; a *flag* value to control the use of memory by the C program; two arrays to provide the C program's heap, static and stack space; and a pair of channels for passing channel pointers to the C program.

Procedural interface

The Type 2 occam interface is defined as follows:

```
PROC PROC.ENTRY (CHAN OF SP fs, ts,
                 VAL INT flag,
                 []INT ws1, ws2,
                 []INT in, out)
```

The parameters are described below:

- `fs` — a channel from the host file server to the C program.
- `ts` — a channel from the program to the host file server.

  The channels `fs` and `ts` are connected to the channels `in[1]` and `out[1]` which are passed as parameters to the C program — these are provided for the use of the C runtime libraries only, and should not be used by the application code.

- `flag` — indicates whether one or two workspaces are to be used.

  If the value of `flag` is set to 0 then the program will run with two workspace areas; one for static and heap data, the other for the runtime stack. If the value of `flag` is set to 1 then the program will run with a single combined workspace.

- `ws1` — used by the C program for its workspace.

  If `flag` is 0 then this array is used only for the runtime stack, if `flag` is 1 then it is used as the program's combined workspace (static, heap *and* stack).

- `ws2` — used by the C program as its static/heap workspace when `flag` is set to zero, otherwise unused.

- `in` — an array of pointers to occam channels going to the C program.

- `out` — an array of pointers to occam channels going from the C program.

Note: The first two elements in the channel pointer arrays `in` and `out` are reserved for use by the C program's runtime system and cannot be used by the program.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to the number of elements in the occam arrays `in` and `out`
- `in[0]` and `out[0]` are set to NULL
- `in[1]` is a pointer to the `fs` channel and is used by the C runtime system to communicate with the host
- `out[1]` is a pointer to the `ts` channel and is used by the C runtime system to communicate with the host
- The remaining elements of the arrays `in` and `out` are set to the values in the corresponding elements of the occam arrays

Example

The following example is an occam procedure, `call.prog2`, which calls a C program via the `PROC.ENTRY` procedure interface:

```
#INCLUDE "hostio.inc"

PROC call.prog2 (CHAN OF SP fs, ts,
                 CHAN OF COMM to.process,
                 CHAN OF COMM from.process)

  #USE "hostio.lib"
  #USE "centry.lib"       -- C interface code

  VAL flag IS 1 :         -- combined heap and stack
  [100000]INT ws1 :       -- stack and heap for program
  [1]INT ws2 :            -- dummy workspace for program
  [3]INT in, out :        -- channel pointers (not used)

  SEQ
    -- set up user output channel
    LOAD.OUTPUT.CHANNEL(out[2], from.process)

    -- set up user input channel
    LOAD.INPUT.CHANNEL(in[2], to.process)

    -- call program
    PROC.ENTRY(fs, ts, flag, ws1, ws2, in, out)
    so.exit(fs, ts, sps.success)
:
```

Two channels are declared of type `COMM`, the first being an input channel to the process, the second an output channel from the process. (The declaration of protocol type `COMM` is assumed.)

10.2.6 Type 3 interface definition

The Type 3 interface is used to run programs which communicate with other processes on the same processor or in a network of processes, but which do not require access to host services. Processes built with the Type 3 interface can communicate with other processes through channels in the same way as Type 2 processes.

Programs using the Type 3 interface *must* be linked with the reduced C runtime library.

The parameters for the Type 3 procedure are: a *flag* value to control the use of memory by the C program; two arrays to provide the C program's heap, static and stack space; and a pair of channels for passing channel pointers to the C program.

Procedural interface

The interface for Type 3 equivalent occam processes is defined below:

```
PROC PROC.ENTRY.RC (VAL INT flag,
                    []INT ws1, ws2,
                    []INT in, out)
```

The parameters are described in the following list.

- `flag` — indicates whether one or two workspaces are to be used.

  If the value of `flag` is set to 0 then the program will run with two workspace areas; one for static and heap data, the other for the runtime stack. If the value of `flag` is set to 1 then the program will run with a single combined workspace.

- `ws1` — used by the C program for its workspace.

  If `flag` is 0 then this array is used only for the runtime stack, if `flag` is 1 then it is used as the program's combined workspace (static, heap *and* stack).

- `ws2` — used by the C program as its static/heap workspace when `flag` is set to zero, otherwise unused.

- `in` — an array of pointers to occam channels going to the process.

- `out` — an array of pointers to occam channels coming from the process.

Note: The first two elements in the channel pointer arrays `in` and `out` are reserved for use by the C program's runtime system and cannot be used by the occam program.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to the number of elements in the occam arrays `in` and `out`

- `in[0]`, `in[1]`, `out[0]` and `out[1]` are are set to NULL

- The remaining elements of the arrays `in` and `out` are set to the values in the corresponding elements of the occam arrays

Example

The following shows how to call a Type 3 equivalent occam process from occam source, and how to set up the parameters required. The example consists of an occam procedure 'call.prog3' within which a C program is called.

```
PROC call.prog3 (CHAN OF COMM to.process,
                 CHAN OF COMM from.process)

  #USE "centry.lib"      -- C entry point library

  VAL flag IS 0 :        -- separate heap and stack

  [1000]INT ws1 :        -- stack for program
  [40000]INT ws2 :       -- heap for program
  [3]INT in, out :       -- pointers to inputs/outputs

  SEQ
    -- set up user output channel
    LOAD.OUTPUT.CHANNEL(out[2], from.process)

    -- set up user input channel
    LOAD.INPUT.CHANNEL(in[2], to.process)

    -- call program
    PROC.ENTRY.RC(flag, ws1, ws2, in, out)
:
```

Two channels are declared of type COMM, the first being an input channel to the process, the second an output channel from the process. (The declaration of protocol type COMM is assumed.)

The first statement sets up a pointer to the output channel, using the procedure LOAD.OUTPUT.CHANNEL. The second statement sets up a pointer to the input channel, using the procedure LOAD.INPUT.CHANNEL. Note that the first two input and output channels are reserved by the runtime system even though there is no host communication taking place.

10.2.7 Building the occam equivalent process

The occam equivalent processes built from these interfaces can be called from an occam program in the same way as any other occam procedure. Note that, because the interface procedures have fixed names, there can only be one process of a particular type in each linked unit. However, multiple C programs called in this way may be placed on a processor by the configurer.

Once all the component C and occam code for the complete program has been compiled, it is linked with the C runtime libraries, the occam entry points library and any other occam libraries required. The program is then configured and a bootable code file produced.

The occam interface code is supplied in the library `centry.lib`. The C libraries can be linked by using the linker control file `clibs.lnk`, for the *full* runtime library, or `clibsrd.lnk`, for the *reduced* runtime library. For example, consider a program that consists of the following compiled files:

- `main.tco` — the compiled C program to be called from occam
- `wrap.tco` — the compiled occam code that calls the interface procedure

This program can be linked with the full run-time libraries, for a 32 bit transputer, using the following command:

```
ilink wrap.tco main.tco -f clibs.lnk -f occama.lnk
```
(UNIX toolsets)

```
ilink wrap.tco main.tco /f clibs.lnk /f occama.lnk
```
(MS–DOS/VMS toolsets)

11 EPROM programming

INMOS EPROM software is designed so that programs can be developed and tested using the INMOS toolset, and once they are working, can be placed in ROM with only minor change.

11.1 Introduction

During development, software is booted onto a network from a link connecting the network to the host computer. Then the software is prepared for a ROM, which is attached to the root transputer in the network.

Figure 11.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

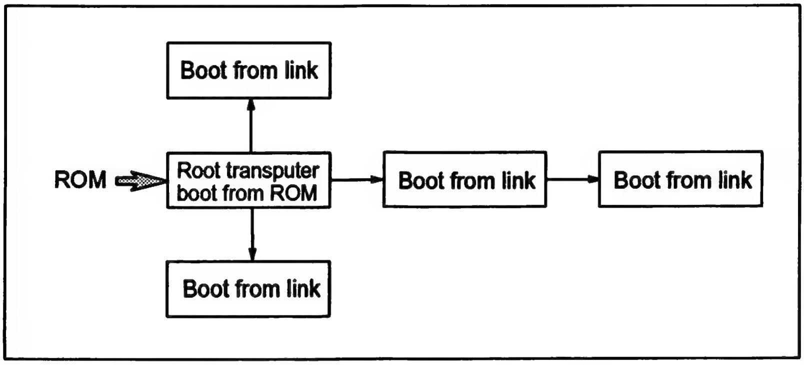


Figure 11.1 Loading a network from ROM

To prepare software to be booted from ROM, rather than to be booted from link, the following two steps must be taken:

1. Give different options to the configurer and collector tools so that they produce ROM-bootable code.
2. Run the `ieprom` tool to produce a file or set of files suitable for blowing into EPROM.

Figures 11.2 and 11.3 illustrate the stages of preparing ROM-bootable software. Figure 11.2 shows an occam program, compiled and linked for a single processor. Figure 11.3 shows a configured program, consisting of one or more linked units, connected together and allocated to processors as described in a configuration file.

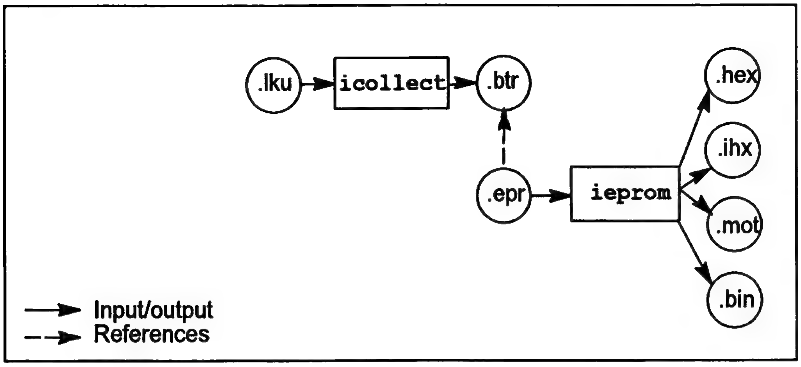


Figure 11.2 Preparation of ROM–bootable software (single occam program)

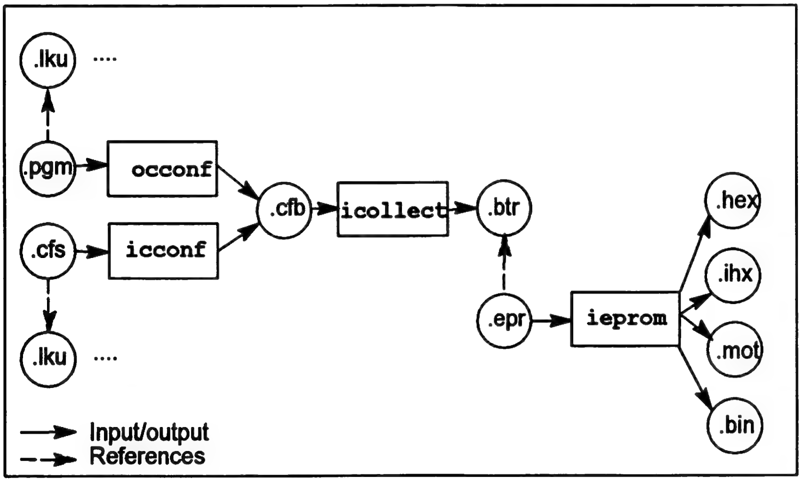


Figure 11.3 Preparation of ROM–bootable software (configured program)

11.2 Processing configurations

The processing configuration used will depend on the number of software processes, the number of transputers available to run the code and whether the code is to run from ROM or RAM. The following sections outline the possible configurations.

When preparing FORTRAN or C code to be booted from ROM the configurer must be used in order to specify the size of stack and heap. This applies even when the application consists of a single process running on a single processor. A single occam process can be configured or prepared as a single, linked program.

11.2.1 Single processor, run from ROM

The application process is prepared as one or more processes, connected as described in a configuration file. If the application consists of a single occam program then it can be prepared without using the configurer. It is then run on a single processor, with the code in ROM, and the RAM is used as the data area.

11.2.2 Single processor, run from RAM

The application process is prepared as one or more processes, connected as described in a configuration file. If the application consists of a single occam program then it can be compiled and linked without using the configurer. When booted from ROM, the processor copies the code into RAM and runs it, using the RAM for the data area.

11.2.3 Multiple process, multiple processor, run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads its own code into RAM, and loads the rest of the network via its links. Each processor then sets off its own processes, and the application runs. (This is the configuration shown in figure 11.1).

11.2.4 Multiple process, multiple processor, root run from ROM, rest of network run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads the rest of the network via its links, and then continues to run its own code from the ROM.

11.3 The EPROM tool: `ieprom`

The EPROM tool `ieprom` takes the output of the collector, and produces a file or set of files suitable for blowing into an EPROM. The following output formats are supported:

- Binary
- Hex
- Intel hex format
- Intel extended hex format
- Motorola S-record format

`ieprom` supports the production of code files in *block mode*, which allows the code to be placed in a set of different files. This is useful to program EPROMS organized as separate byte-wide devices, or where the EPROM programming device does not have enough memory to hold the entire image.

`ieprom` also supports the inclusion in the EPROM image of a *memory configuration*. Some 32-bit transputers have a configurable memory interface which can be initialized from a fixed area in the ROM, when the transputer is reset. A particular memory configuration can be specified to `ieprom` in a text file. These files are known as memory configuration files and normally have the file extension `.mem`. The format of these files, and the facility to edit them using an interactive tool called `iemit` is described in chapter 6 of the *Toolset Reference Manual.*

`ieprom` is driven by a control file which normally has the file extension `.epr`. A detailed description of `ieprom` and its control file is given in chapter 7 of the *Toolset Reference Manual.*

11.4 Using the configurer and collector to produce ROM-bootable code

To produce code suitable for running in ROM or RAM, the configurer and collector tools must be specified with the appropriate command line options. The following options are used to configurer single and multi-processor programs and to collect unconfigured single processor programs:

- The `ro` option specifies that the code is to run in ROM.
- The `ra` option specifies that the code is to run in RAM.
- The `rs` option specifies the ROM size (if not specified in configuration file). This option does not apply to the `occonf`, see below.

In addition, if using `icconf`, the `P` option must be used with the configurer, in order to specify the root processor name.

If using `occonf`, the `NETWORK` description in the configuration file should indicate:

- which processor is the root processor, by setting its `root` attribute to `TRUE`
- the size of the ROM on that processor, by setting its `romsize` attribute to the appropriate value, in bytes.

The collector will add the appropriate ROM bootstrap to the application code and the output file will be given the extension `.btr`.

11.5 Summary of EPROM tool steps for different configurations

11.5.1 Using `icconf`

	Compile and link	Configure	Collect	EPROM
Single processor, run from ROM.	Compile and link a set of units, one per process.	Configure with the `ro`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the `ra`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the `ra`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor root runs from ROM, rest of network runs from RAM.	Compile and link a set of units, one per process.	Configure with the `ro`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

11.5.2 Single processor unconfigured occam program

	Compile and link	Configure	Collect	EPROM
Run from ROM.	Compile and link program.	Not needed.	Collect with the `ro` and `t` options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Run from RAM.	Compile and link program.	Not needed.	Collect with the `ra` and `t` options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

11.5.3 Using `occonf`

	Compile and link	Configure	Collect	EPROM
Single processor, run from ROM.	Compile and link a set of units.	Configure with the `ro` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single processor, run from RAM.	Compile and link as a set of units.	Configure with the `ra` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, run from RAM.	Compile and link a set of units.	Configure with the `ra` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, root runs from ROM, rest of network runs from RAM.	Compile and link a set of units.	Configure with the `ro` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

12 Dynamic code loading

12.1 Introduction

This document describes the dynamic code loading facility provided by the INMOS ANSI C toolset. This facility allows code to be loaded and executed *at runtime*. This can be of use to keep internal memory requirements down. Code might be dynamically loaded, when for example, it is known to be used only rarely, or if it cannot be determined until runtime what code is required. Dynamically loading and executing the code, followed by freeing up the space that it required, circumvents the need to have code (that may never be run or run only rarely) taking up space in internal memory.

Only users who are knowledgeable about the implementation of ANSI C and are familiar with the construction of C runtime systems in general, should attempt to use this facility.

12.2 Overview

Functions are provided in the C library to dynamically load code from:

1 a `.rsc` file,

2 the image of a `.rsc` file in ROM or RAM,

3 a `.rsc` file transmitted over a channel.

A `.rsc` file is the output of the collector when used with the '`t`' and '`k`' options together. The format of a `.rsc` file is described in chapter 3 of the *ANSI C Toolset Reference Manual.*

Within this chapter, the code in the `.rsc` file is referred to as the 'child', and the code that loads and calls the code in the `.rsc` file is referred to as the 'parent'. To the parent the child is a function and is known only by a pointer to that function. The parent can thus call the child either as a parallel process or by de-referencing the pointer to it.

The child can be code that is compiled (for the transputer), linked and collected from source in any programming language. However, each language implementation has its own runtime model requirements. For example, if a static area is present then it will need to be initialized. A `.rsc` file only gives one entry point, as shown in figure 12.1.

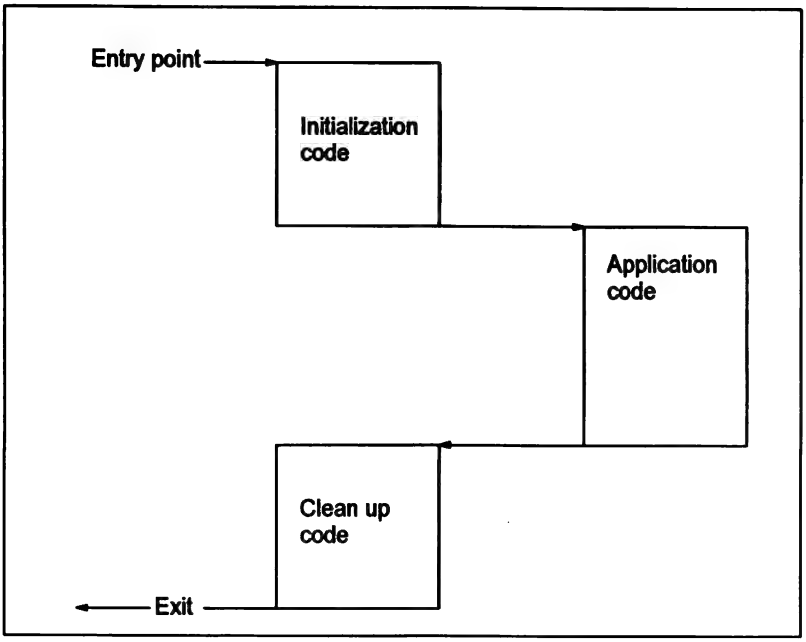


Figure 12.1 Structure of child

For some language implementations it may be that no initialization or clean up need be done. For INMOS ANSI C, initialization and clean up code is needed, but just how much depends on what the application code does. Full initialization and clean up code for INMOS ANSI C is provided in two harness files that have to be edited to be appropriate to the application code.

It is suggested that this chapter is read in conjunction with the supplied source files, `fnload1.c` and `fnload2.c` which are fully commented. The files can be found in the runtime startup system source directory. See the accompanying Delivery Manual for details.

Having the initialization and clean up code under one's own control means that there is no restriction on the interface of the application code, e.g. there need not be a function called `main`.

The memory requirements of the child, e.g. static and heap, must be allocated by the parent. The parent passes over their details to the child via the parent-child interface. When the child has finished its work, the parent can free up these areas. How this is done is explained in the next sections.

12.3 Basic Parenthood

The parent will have to obtain some details of the child from the `.rsc` file so that it can take appropriate action. It obtains this information by calling one of the following functions, depending on where the `.rsc` file is found:

- `get_code_details_from_file`
- `get_code_details_from_memory`
- `get_code_details_from_channel`

The specifications of these functions are given in chapter 2 of the *ANSI C Language and Libraries Reference Manual.*

Each of these functions takes as one of their parameters a pointer to a structure that will be initialized with details of the child, as long as the function succeeds. The structure, given in the header file `fnload.h`, is:

```
struct fn_data
  {
    int    target_processor_type; /* as given in the .rsc file */
    size_t stack_size;            /* in bytes */
    size_t vectorspace_size;      /* in bytes */
    size_t static_size;           /* in bytes */
    size_t entry_point_offset;    /* in bytes */
    size_t code_size;             /* in bytes */
  };
typedef struct fn_data fn_info;
```

where:

> `target_processor_type` gives the processor type for which the code in the `.rsc` file is compiled. The processor type is encoded as an integer the values of which are given in chapter 3 of the *ANSI C Toolset Reference Manual.*
>
> `stack_size` gives the required size in bytes of the stack. **Note:** for some languages, e.g. C, it may not be possible for this value to be calculated. Hence one may not always be able to rely on this value. Check with the documentation associated with the implementation of the tools used to generate the `.rsc` file. (e.g. In the INMOS occam toolsets Dx205 and Dx305, the occam compiler calculates the size of the stack exactly).
>
> `vectorspace_size` gives the required size in bytes of the vectorspace. If the child is written in a programming language which does not use vectorspace then this value will be zero.
>
> `static_size` gives the required size in bytes of the static area. If the child is written in a programming language which does not use static data then this value will be zero.

`entry_point_offset` is the size (in bytes) of the difference in position, of the entry point of the code and the start of the code block in the `.rsc` file. This value is used by the `load_code_` functions mentioned below.

`code_size` is the size in bytes of the code block in the `.rsc` file.

The parent will need to claim enough space for the child's needs using, say, `malloc` or part of a static array. The parent, however, will need to know more about the child than is given in a `.rsc` file. If the child is written in ANSI C then heap may be required, and no heap size value is given in the `.rsc` file. Further, the stack value given in the `.rsc` file will not, in general, be exact:

- because C has dynamic memory allocation, the amount of heap required cannot be known in advance, and as such the parent will have to estimate how much space to claim for it;
- because C has recursion the amount of stack required cannot be known in advance. However, if the child is not called as a parallel process then it will share its parent's stack, and so in this case the parent need not set aside specific stack space for the child. If, on the other hand, the child is called as a parallel process, then it will run in its own stack, the size of which, as with the heap size, must be estimated.

For the parent to load the child into the code space that has been set aside for it (of size `code_size` bytes), it calls one of the following functions, depending on where the `.rsc` file is found:

- `load_code_from_file`
- `load_code_from_memory`
- `load_code_from_channel`

The specifications of these functions are given in chapter 2 of the *ANSI C Language and Libraries Reference Manual.*

These functions return a pointer to the start of the child, in the form of a pointer to a function taking no parameters and returning `void`. If the child has been set up to run as a parallel process then this pointer can be used directly in `ProcInit` or `ProcAlloc`. Otherwise, this pointer will need to be cast into a pointer to a function of the type of the child's interface before being called with appropriate parameters.

There is one case in which the `load_code_from_memory` function may not need to be called. The `get_code_details_from_memory` function returns a pointer to the start of the child, though, in contrast to the `load_code_` functions, the pointer is returned via the parameter list. If it is possible to run the child where it is in memory then there is no need to waste time by calling the function `load_code_from_memory`. This function simply copies the code block of the `.rsc` file from one part of memory to another. Where `load_code_from_memory` would be of use is when the `.rsc` file is in ROM and it is desired to run the code from RAM for performance reasons.

Before we go into further details about parents, it would be best to see an actual example of dynamic code loading in action. The child is written in INMOS ANSI C, and is chosen to be as simple as possible because we have not yet discussed the details of the C harness files for children. The source code for the child is as follows:

```
/*
 *  This example function does not require any static, heap or i/o.
 *  Hence it needs no harness, but the descriptor pragma is, as
 *  always, necessary.
 */

#pragma IMS_off(stack_checking)

int add3( int );
#pragma IMS_descriptor( add3, ansi_c, 0, 0, "" )


int add3( int i )
{
  return( i + 3 );
}
```

The pragmas are explained as below, details of all INMOS C compiler pragmas are given in chapter 1 of the *ANSI C Toolset Reference Manual.*

Without setting up stack checking variables in the static area, stack checking will not work for the child and so the `IMS_off` pragma is included for safety as it will override selecting stack checking on the compiler command line. Details of setting up stack checking are given in section 12.5.2. The `IMS_descriptor` pragma is necessary in all children written in INMOS ANSI C for implementation reasons. It is described in more detail in section 12.4.1.

If the child's source code is in a file called `add3.c`, then to turn the child into a `.rsc` file, the following commands should be used, if the code is to run on a TA class transputer:

UNIX based toolsets:

```
icc add3.c -ta
ilink add3.tco -ta -me add3
icollect add3.lku -t -k
```

MS-DOS/VMS based toolsets:

```
icc add3.c /ta
ilink add3.tco /ta /me add3
icollect add3.lku /t /k
```

(Options for specifying other transputer types, are listed in appendix B of the *ANSI C Toolset Reference Manual).*

Now for the parent. We know that the child does not need static, heap, vectorspace or input/output, so the parent need only set aside enough space for the code of the child. Assume that the `.rsc` file resides on disc:

```
/*
 *  Example program which dynamically loads from a file a simple
 *  function that adds 3 to its parameter; the function does not
 *  require any static, heap or i/o.
 *  The function is expected already to be in a .rsc file called
 *  "add3.rsc".
 */


#include <stdio.h>
#include <stdlib.h>
#include <fnload.h>

int main( void )
{
  char* rsc_filename = "add3.rsc";
  fn_info file_details;
  size_t header_size;

  printf( "Main program started.\n" );
  if (get_code_details_from_file( rsc_filename,
                                  &file_details, &header_size ) != 0)
    printf(
     "Details were not successfully retrieved from the .rsc file.\n");
  else
    {
      void* code_area_base;

      if ((code_area_base = malloc(file_details.code_size)) == NULL)
        printf("Malloc failed to allocate enough space for code.\n");
      else
        {
          loaded_fn_ptr fn_ptr;

          if ((fn_ptr = load_code_from_file(rsc_filename,
                                            &file_details,
                                            header_size,
                                            code_area_base)) == NULL)
            printf(
             "Code was not successfully loaded from the .rsc file.\n");
          else
            {
              typedef int (*loaded_code_fn_ptr)(int);

              if ( ( *((loaded_code_fn_ptr)fn_ptr) )(71) == 74 )
                printf( "\n***Dynamically loaded code gave correct "
                        "value. ***\n\n" );
              else
                printf( "\n<<<Dynamically loaded code gave wrong "
                        "answer.>>>\n\n" );
            }
          free( code_area_base );
        }
    }
  printf( "Ending main program.\n" );
  return 0;
}
```

The parent had to know the interface of the child so that it could type cast `fn_ptr` to the correct type for correct calling of the child.

Before going on to look at more complicated parents, we need to become more familiar with children.

12.4 Childhood in INMOS ANSI C

The discussion in this section is specific to the INMOS ANSI C toolset, Dx314.

To attend to necessary initializations and tidying up, children will, in general, require two harness files. (There are two files rather than one because there is no other way to bootstrap the global static base (gsb).) These two files are provided for the user to edit as appropriate to their application code.

The harness files are very similar to those used in the modifiable C runtime startup system, see chapter 3 of the *ANSI C Language and Libraries Reference Manual*. This is no coincidence – the child is basically a full C program, except that it need not start with a function called `main`. It is worth keeping in mind the idea that the child is a program in its own right when determining what some functions do when called in the child. For example, `exit` and `abort` return control to the parent, and `max_stack_usage` refers to the child, only if called by the child, and refers to the parent, only if called by the parent. However, the parent and the child can share the facilities of the host, and use of common facilities must be performed with care by the programmer as there is no built-in protection.

There are two editing tasks to attend to in the harness files. One is to ensure that the harness interfaces properly with the application code. The parameters of, and possibly the return value from, the application code's interface, need to be connected from the parent to the application through the two harness files. The other task is to edit or comment out unused parts of the harness.

For example, if the function `clock` is not called by the application code then there is no need to leave the lines:

```
    _IMS_clock_priority = ProcGetPriority();
    _IMS_StartTime = (clock_t)ProcTime();
```

in the second file of the harness. Removing these lines would reduce the size and increase the speed of the child. There is, however, no harm in leaving the lines in, even if it is known that they are superfluous.

When library routines are used it is sometimes not clear when heap or static is required, so here are some simple guidelines:

- input/output (i/o) and concurrency require heap;
- use of anything in the second file of the harness requires static. (Hence, as heap requires static, i/o and concurrency require static too.)

The two harness files are highly commented to indicate when a statement is or is not required. For further details about the meaning of the variables and functions used in the harness files see chapter 3 of the *ANSI C Language and Libraries Reference Manual.*

The procedure for obtaining a `.rsc` file for the child is always compile, link, collect. The system is not set up for the configurer to be used on a child. Linking should be done with the linker indirect file `clibs.lnk` or, for a reduced system, `clibsrd.lnk`, and the main entry point identifier must be given using the '`me`' option. Collecting should be done with the 't' and 'k' options together.

12.4.1 The `IMS_descriptor` pragma

In the first file of a child there must appear the pragma `IMS_descriptor`. The need for it stems from the implementation of INMOS ANSI C. Not all of its functionality is required for dynamic code loading; only its first two parameters really matter.

The first parameter is the identifier of the function which is to be the entry point of the `.rsc` file. The second parameter should be `ansi_c`.

The third parameter is the amount of workspace required by the child, in words. This value becomes `stack_size` in the structure that the `get_code_details_` functions return, after having been transformed into the amount in bytes. (Use of the '`s`' option on the collector allows you to add an amount to this value.) If this value can be worked out then putting it as this third parameter would save the parent estimating the value; if it cannot be worked out, then specify zero for this parameter and leave the parent to estimate the requirement.

The fourth parameter is the amount of vectorspace required by the child, in words. Vectorspace is not used by INMOS ANSI C and so should always be zero.

The last parameter is a string which serves no purpose in dynamic code loading and so should be set to the empty string: "".

Further details of the `IMS_descriptor` pragma can be found in chapter 1 of the *ANSI C Toolset Reference Manual.*

12.5 Advanced Parenthood

Let us now look at what to do in a parent whose child requires static, heap, i/o, stack checking and the ability to call the functions `max_stack_usage` and `get_details_of_free_stack_space`.

We choose a specific child in INMOS ANSI C to clarify the discussion. The child consists of three files, the first two of which are edited versions of the source harness files. The third file is the application code which, for this example, simply causes stack overflow. The files of the child are shown below:

stack1.c

```
/*
 *  This file contains the first stage of the dynamic loading of a C
 *  function which requires static, heap, i/o, stack checking and
 *  to be able to call max_stack_usage().
 *
 *  This source is used to build the entry point c_fnload_stage1.
 *
 *  STAGE 1 : a) initialise the static area.
 *            b) call stage 2 and bootstrap the gsb.
 *
 */


#pragma IMS_off(stack_checking)

#include <channel.h>  /* for type Channel */
#include <stddef.h>   /* for size_t */


#pragma IMS_translate(initialise_static, "initialise_static%c")
extern int initialise_static(void *hidden_gsb, char *area_start,
                             unsigned int area_size,
                             unsigned int available_size);
#pragma IMS_nolink(initialise_static)


extern int c_fnload_stage2(void* hidden_gsb,
                           void* stack_addr,
                           size_t stack_size_in_bytes,
                           void* heap_addr,
                           size_t heap_size_in_bytes,
                           Channel* in,        Channel* out );
#pragma IMS_nolink(c_fnload_stage2)


int c_fnload_stage1( void* static_addr, size_t static_size_in_bytes,
                     void* stack_addr,  size_t stack_size_in_bytes,
                     void* heap_addr,   size_t heap_size_in_bytes,
                     Channel* in,       Channel* out );
#pragma IMS_descriptor(c_fnload_stage1, ansi_c, 0, 0, "" )


int c_fnload_stage1( void* static_addr, size_t static_size_in_bytes,
                     void* stack_addr,  size_t stack_size_in_bytes,
                     void* heap_addr,   size_t heap_size_in_bytes,
                     Channel* in,       Channel* out )
{
  initialise_static(static_addr, (char *)static_addr,
                    (unsigned int)static_size_in_bytes,
                    (unsigned int)static_size_in_bytes );

  return c_fnload_stage2( static_addr,
                          stack_addr,  stack_size_in_bytes,
                          heap_addr,   heap_size_in_bytes,
                          in,          out );
}
```

stack2.c

```
/*
 *  This file contains the second stage of the dynamic loading of a
 *  C function which requires static, heap, i/o, stack checking and
 *  to be able to call max_stack_usage().
 *
 *  This source is used to build the entry point c_fnload_stage2.
 *
 *  Note that this code relies on the presence of a static area.
 *
 */


#pragma IMS_off(stack_checking)

#include <setjmp.h>      /* for setjmp  */
#include <channel.h>     /* for Channel */
#include <stdlib.h>      /* for exit    */
#include "uglobal.h"     /* for globals */
#include "startup.h"     /* for set_host_link and io_and_hostinfo_init
*/


int stack_overflow(void);

int c_fnload_stage2( void* stack_addr,  size_t stack_size_in_bytes,
                     void* heap_addr,   size_t heap_size_in_bytes,
                     Channel* in,       Channel* out )
{
  _IMS_stack_limit        = (int *)stack_addr;
  _IMS_stack_base         = (int *)((size_t)stack_addr +
stack_size_in_bytes);

  _IMS_heap_start         = (int *)heap_addr;
  _IMS_heap_init_implicit = TRUE;
  _IMS_heap_size          = (unsigned int)heap_size_in_bytes;
  _IMS_sbrk_alloc_request = SBRK_REQUEST;


  set_host_link(in, out);
  io_and_hostinfo_init();


  if (setjmp(_IMS_startenv) == 0)
    {
      exit(stack_overflow());
    }
  return _IMS_retval;
}
```

stack3.c

```
/*
 *  Third stage of example of overflowing the stack and
 *  having this reported by the runtime library.
 */


/* ensure that stack checking is switched on */
#pragma IMS_on( stack_checking )


#include <stdio.h>
#include <misc.h>

void foo( int a )
{
  char stack_eating_array[ 512 ];

  stack_eating_array[4] = 's';
  printf( "max_stack_usage() gives %ld bytes\n", max_stack_usage() );
  foo (a);
}


int stack_overflow( void )
{
  printf( "stack_overflow: started\n" );

  foo (42);

  printf( "stack_overflow: finished\n" );
  return 0;
}
```

To obtain a `.rsc` file from these files for a TA class transputer use the following commands:

UNIX based toolsets:

```
icc stack1.c -ta
icc stack2.c -ta
icc stack3.c -ta -ks
ilink -ta -f child.lnk
icollect stack1.lku -t -k
```

MS-DOS/VMS based toolsets:

```
icc stack1.c /ta
icc stack2.c /ta
icc stack3.c -ta /ks
ilink /ta /f child.lnk
icollect stack1.lku /t /k
```

where `child.lnk` is:

```
#mainentry c_fnload_stage1
#include clibs.lnk
stack1.tco
stack2.tco
stack3.tco
```

Note: that we have to give the main entry point identifier to the linker and that we use the linker indirect file `clibs.lnk`. If the child did not require the server on the host then we would use the linker indirect file `clibsrd.lnk`.

From the interface of the function in `stack1.c` above, one sees that the parent must pass static, stack, heap and i/o details to the child. The methods by which the parent obtains these details follows:

12.5.1 static

A call to a `get_code_details_` function will give details of how much static is required and this value can be used directly in `malloc`. The pointer returned by the call to `malloc` is passed for `static_addr` and the amount of static that was claimed in the call to `malloc` is passed for `static_size_in_bytes`.

12.5.2 stack

The stack formal parameters are required if any of the following are desired in the child: stack checking, parallel processing, the ability to call `max_stack_usage` or `get_details_of_free_stack_space`.

If the child is not going to be called as a parallel process then the function `get_details_of_free_stack_space`, will provide values that can be passed for `stack_addr` and `stack_size_in_bytes`.

If the child is going to be called as a parallel process and if the stack formal parameters are required, then `ProcInit` rather than `ProcAlloc` must be used. Because for `ProcInit` you must provide a stack (which would typically be obtained by using an estimated value for the size of the stack in a call to `malloc`), you would therefore have the values ready to pass over for `stack_addr` and `stack_size_in_bytes`.

(Note: that if you do provide this separate stack, then for word alignment its size should be such that `(stack_size_in_bytes % sizeof(int)) == 0`.) If the stack formal parameters are not required, then both `ProcAlloc` and `ProcInit` are valid for use.

12.5.3 heap

The amount of heap required must be estimated. This amount can then be used in a call to `malloc`. The pointer that is returned from the call to `malloc` is passed for `heap_addr`, and the amount of heap that was requested in the call to `malloc` is passed for `heap_size_in_bytes`.

12.5.4 input/output

For the child to communicate with the server running on the host it must be given channels to transmit its requests over. There are two functions in the C library provided, that the parent can call to determine what these channels are:

- `from_host_link()`: returns a pointer suitable to be passed for `in`.

- `to_host_link()`: returns a pointer suitable to be passed for `out`.

Alternatively, the parent may pass to the child, channels which first go through a multiplexor before reaching the server. This latter situation would be desirable if the parent were to dynamically load several children all needing to communicate with the server.

We can now give the source of a parent which calls the above child (sequentially):

stack.c

```
/*
 *  Example program which dynamically loads code from a file where
 *  that code is to be stack checked and use max_stack_usage(). The
 *  dynamically loaded code is called directly from the main program,
 *  rather than from a parallel process or as a parallel process.
 *  The dynamically loaded code requires static, heap, i/o,
 *  stack checking and to call max_stack_usage().
 *
 *  The child code is expected already to be in a .rsc file called
 *  "stack1.rsc".
 */


#include <stdio.h>
#include <stdlib.h>
#include <fnload.h>
#include <hostlink.h>
#include <misc.h>
#define CHILD_HEAP_SIZE_IN_BYTES  (size_t)9216   /* 9K */


int main( void )
{
  char* rsc_filename = "stack1.rsc";
  fn_info file_details;
  size_t header_size;

  printf( "Main program started.\n" );
  if (get_code_details_from_file( rsc_filename, &file_details,
                                                 &header_size ) != 0)
    printf(
      "Details were not successfully retrieved from the .rsc file.\n");
  else
    {
      void* code_area_base;

      if ((code_area_base = malloc( file_details.code_size )) == NULL)
        printf( "Malloc failed to allocate enough space for code.\n" );
      else
        {
          void* static_area_base;

          if ((static_area_base = malloc(file_details.static_size))
                                                              == NULL)
            printf(
               "Malloc failed to allocate enough space for static.\n");
          else
            {
```

```
            void* heap_area_base;

            if( (heap_area_base = malloc(CHILD_HEAP_SIZE_IN_BYTES))
                                                            == NULL)
              printf( "Malloc failed to allocate enough"
                      " space for the heap.\n" );
            else
              {
                loaded_fn_ptr fn_ptr;

                if ( (fn_ptr = load_code_from_file( rsc_filename,
                                                    &file_details,
                                                    header_size,
                                                    code_area_base))
                                                    == NULL )
                  printf( "Code was not successfully loaded"
                          " from the .rsc file.\n" );
                else
                  {
                    void *child_stack_addr;
                    size_t child_stack_size_in_bytes;
                    typedef int (*loaded_code_fn_ptr)(void*, size_t,
                                                      void*, size_t,
                                                      void*, size_t,
                                                      Channel*,
                                                      Channel*);

                   get_details_of_free_stack_space(&child_stack_addr,
                                             &child_stack_size_in_bytes);

                    /*
                     *       call the dynamically loaded code
                     */
                    (void)( *((loaded_code_fn_ptr)fn_ptr) )
                             (static_area_base,
                              file_details.static_size,
                              child_stack_addr,
                              child_stack_size_in_bytes,
                              heap_area_base,
                              CHILD_HEAP_SIZE_IN_BYTES,
                              from_host_link(),
                              to_host_link() );
                  }
                free( heap_area_base );
              }
            free( static_area_base );
          }
        free( code_area_base );
      }
    }
  printf( "Ending main program.\n" );
  return 0;
}
```

The commands to generate a bootable for the parent are standard. For example, to generate a bootable for a TA class transputer, use the following commands:

UNIX based toolsets:

```
icc stack.c -ta -o stack.tah
ilink -f stack.lnk -ta -o stack.cah
icollect stack.cah -t
```

MS-DOS/VMS based toolsets:

```
icc stack.c /ta /o stack.tah
ilink /f stack.lnk /ta /o stack.cah
icollect stack.cah /t
```

where `stack.lnk` is:

```
stack.tah
#include cnonconf.lnk
```

Alternatively, as long as the `stack.lnk` file exists, the tool `imakef` can be used to generate a makefile for the parent:

```
imakef stack.bah -c
```

If we wanted to set the child off as a parallel process, then the parent and the first file of the child would have to change slightly – the parent uses `ProcInit` and the child has a `Process*` parameter as its first parameter (and the child also ensures that each of its parameters occupies one word to make the call to `ProcInit` simpler):

stack1.c

```
/*
 *  This file contains the first stage of the dynamic loading of a C
 *  function which requires static, heap, i/o, stack checking and
 *  to be able to call max_stack_usage().
 *
 *  This source is used to build the entry point c_fnload_stage1, and
 *  is suitable for calling as a parallel process.
 *
 *  STAGE 1 : a) initialise the static area.
 *            b) call stage 2 and bootstrap the gsb.
 */


#pragma IMS_off(stack_checking)

#include <process.h>  /* for Process */
#include <channel.h>  /* for type Channel */
#include <stddef.h>   /* for size_t */


#pragma IMS_translate(initialise_static, "initialise_static%c")
extern int initialise_static(void *hidden_gsb, char *area_start,
                             unsigned int area_size,
                             unsigned int available_size);
#pragma IMS_nolink(initialise_static)


extern int c_fnload_stage2(void* hidden_gsb,
                           void* stack_addr,
                                        size_t stack_size_in_bytes,
                                        void* heap_addr,
                                        size_t heap_size_in_bytes,
                                        Channel* in,
                                        Channel* out);
#pragma IMS_nolink(c_fnload_stage2)


void c_fnload_stage1( Process* p,
                      void* stack_addr,
                      size_t* stack_size_in_bytes_ptr,
                      void* static_addr,
                      size_t* static_size_in_bytes_ptr,
                      void* heap_addr,
                      size_t* heap_size_in_bytes_ptr,
                      Channel* in,
                      Channel* out );
#pragma IMS_descriptor(c_fnload_stage1, ansi_c, 0, 0, "" )


void c_fnload_stage1( Process* p,
                      void* stack_addr,
                      size_t* stack_size_in_bytes_ptr,
                      void* static_addr,
                      size_t* static_size_in_bytes_ptr,
                      void* heap_addr,
                      size_t* heap_size_in_bytes_ptr,
                      Channel* in, Channel* out )
{
  p = p; /* to prevent compiler warning of unused variable */

  initialise_static(static_addr, (char *)static_addr,
                    (unsigned int)*static_size_in_bytes_ptr,
                    (unsigned int)*static_size_in_bytes_ptr );

  (void)c_fnload_stage2( static_addr,
                         stack_addr,   *stack_size_in_bytes_ptr,
                         heap_addr,    *heap_size_in_bytes_ptr,
                         in, out );
}
```

stack.c

```
/*
 *  Example program which dynamically loads code from a file.
 *  The child code requires static, heap, i/o, stack checking and to
 *  call max_stack_usage().
 *
 *  The child is called as a parallel process, thus not needing a cast
 *  of the loaded_fn_ptr variable returned by load_code_from_file.
 *
 *  The child code is expected already to be in a .rsc file called
 *  "stack1.rsc".
 *
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <process.h>
#include <channel.h>
#include <misc.h>
#include <fnload.h>
#include <hostlink.h>


int main( void )
{
  char* rsc_filename = "stack1.rsc";
  fn_info file_details;
  size_t header_size;
  const size_t child_heap_size_in_bytes = 4096;   /* 4K */
  const size_t child_stack_size_in_bytes = 4096;   /* 4K */

  printf( "Main program started.\n" );
  if (get_code_details_from_file( rsc_filename, &file_details,
                                  &header_size ) != 0)
    printf(
    "Details were not successfully retrieved from the .rsc file.\n");
  else
    {
      void* code_area_base;

      if ( (code_area_base = malloc( file_details.code_size ))
           == NULL )
        printf("Malloc failed to allocate enough space for code.\n");
      else
        {
          void* static_area_base;

          if ( (static_area_base = malloc( file_details.static_size ))
               == NULL )
            printf(
             "Malloc failed to allocate enough space for static.\n" );
          else
            {
              void* heap_area_base;

              if((heap_area_base = malloc( child_heap_size_in_bytes))
                   == NULL )
                printf( "Malloc failed to allocate enough space for"
              "the heap.\n" );
              else
                {
                  void* child_stack;

                  if ((child_stack = malloc(child_stack_size_in_bytes))
                       == NULL )
                    printf( "Malloc failed to allocate enough"
                            " space for the stack.\n" );
                  else
                    {
                      loaded_fn_ptr fn_ptr;

                      if ((fn_ptr = load_code_from_file(rsc_filename,
                                                     &file_details,
                                                     header_size,
                                                    code_area_base))
```

```
                                      == NULL )
                                  printf( "Code was not successfully loaded"
                                          " from the .rsc file.\n" );
                                else
                                  {
                                    Process* dynamic_process;

                                    if ( (dynamic_process =
                                          (Process*)malloc(sizeof(Process)))
                                         == NULL )
                                      printf( "Malloc failed to allocate enough"
                                              " space for the process.\n" );
                                    else
                                      if ( ProcInit( dynamic_process,
                                                     fn_ptr,
                                                     child_stack,
                                                     child_stack_size_in_bytes,
                                                     8,
                                                     child_stack,
                                                     &child_stack_size_in_bytes,
                                                     static_area_base,
                                                     &file_details.static_size,
                                                     heap_area_base,
                                                     &child_heap_size_in_bytes,
                                                     from_host_link(),
                                                     to_host_link() ) != 0 )
                                        printf(
                                             "Could not initialise process.\n");
                                      else
                                        {
                                          ProcRun( dynamic_process );
                                          ProcJoin( dynamic_process, NULL );
                                        }
                                  }
                                free( child_stack );
                              }
                            free( heap_area_base );
                          }
                        free( static_area_base );
                      }
                    free( code_area_base );
                  }
              }
  printf( "Ending main program.\n" );
  return 0;
}
```

Although the examples have used the `get_code_details_from_file` and `load_code_from_file` functions, the principles are the same for the cases of the `.rsc` file residing in memory or being transmitted over a channel.

12.6 Childhood in INMOS occam 2

The discussion in this section is specific to the child being written with the INMOS occam 2 (TCOFF) toolset.

occam does not use static or heap areas and so the runtime startup is much simpler than that of C. Also, the occam compiler calculates the exact amount of stack and vectorspace required, so there is no estimating involved in claiming space for the child's needs.

No harness files are required by the occam system, but attention must be given to the interface of the entry point. Firstly, a pointer to the base of the vectorspace area is expected as a hidden parameter after the last visible parameter of the interface. Secondly, occam does not use a static area and so an occam subroutine does not expect a gsb. This means that either a dummy parameter of type `VAL INT` has to be declared in the interface to absorb the gsb that the C system will pass, or the parent uses the provided function `call_without_gsb` to call the child, in which case no dummy parameter is necessary in the occam interface.

To be specific, for an occam interface of

```
PROC occam.program( VAL INT dummy.gsb, CHAN OF SP fs, ts )
```

the call from INMOS ANSI C would be

```
#include <channel.h>
#include <hostlink.h>

typedef void (*loaded_code_fn_ptr)(Channel*, Channel*, void*);
( *((loaded_code_fn_ptr)fn_ptr) )(from_host_link(),
                                  to_host_link(),
                                  vectorspace_area_base);
```

where `vectorspace_area_base` was initialized by, say, a call to `malloc`, and `fn_ptr` was initialized by a call to one of the `load_code_` functions or to `get_code_details_from_memory`.

Alternatively, for an occam interface of:

```
PROC occam.program( CHAN OF SP fs, ts )
```

the call from INMOS ANSI C would be:

```
#include <hostlink.h>

call_without_gsb(fn_ptr,
                 3,
                 from_host_link(),
                 to_host_link(),
                 vectorspace_area_base);
```

There are some further details that the programmer should be aware of when attempting this form of mixed language programming.

If the occam code is required to be run as a parallel process then the interface of the occam code should declare a dummy `VAL INT` parameter after the dummy gsb parameter, or as first parameter if it is known that the gsb will not be passed. This `VAL INT` parameter performs the function of the `Process*` parameter in C functions that are to be run as parallel processes.

If the occam will not be called as a parallel process then it will share the stack of its parent. It is up to the programmer to ensure that there is enough stack left before calling the child. This could be done by comparing the required stack size, as given in the `.rsc` file, with the amount of stack left, as given by the function `get_details_of_free_stack_space`.

A call to the occam procedure `so.exit` terminates the server. Typically it is highly unlikely that the child would ever want to bypass the parent and terminate the server.

For C and occam to pass values between them, we must be sure that the types of the parameters correspond. This and the calling conventions place restrictions on what forms of interface can be used:

- type mapping is given in chapter 10 of this manual;
- only occam functions returning a single value, and occam procedures may be called;
- non-VAL occam parameters should be passed as pointers from C;
- where the formal parameter to an occam procedure or function is an array (VAL or non-VAL) the calling C program should always pass a pointer to the array. For an occam array parameter with unspecified array bounds, the actual sizes of the bounds should be passed immediately following the array parameter; for multidimensional arrays the bounds should be passed in the same order as they appear in the declaration.

Appendices

A Transputer instruction set

This appendix provides a reference for the transputer instruction set as supported by assembly code. For a detailed specification of each of the instructions available, refer to *'Transputer instruction set: a compiler writer's guide'*.

A.1 Prefixing instructions

The transputer instruction set is built up from 16 *direct* instructions, each with a 4-bit argument field. The direct instructions include *prefix* instructions which augment the 4-bit field in a direct instruction which follows them by their own 4-bit argument field, effectively allowing the argument to be extended to 32 bits. Normally, the assembler will compute the prefix instructions required for operand values greater than 4 bits automatically.

pfix	prefix
nfix	negative prefix

A.2 Direct instructions

The direct instructions form the core of the transputer instruction set. Each direct instruction has a single operand, normally an integer constant, which will be encoded in the instruction itself and, if it is larger than will fit into the 4-bit argument field of the direct instruction, into a series of *pfix* and *nfix* instructions as well.

The transputer architecture is based around a three-register *evaluation stack* and a single base register **Wreg**. The load and store 'local' instructions access a word in memory at a displacement from **Wreg** given by the operand value used. The displacement is scaled by the word size. The load and store 'non-local' instructions use the top evaluation stack register (**Areg**) as the base instead of **Wreg**, allowing computed base addresses to be used.

The operand of the *j*, *cj* and *call* instructions is interpreted as a byte displacement from the instruction pointer (program counter) register **Iptr**. *ldpi* is similar but takes its operand from **Areg**.

adc	Add constant operand value to **Areg**.
ajw	Adjust workspace pointer **Wreg** by constant operand value (scaled by word length).
call	Call.
cj	Conditional jump i.e. 'jump if zero otherwise pop **Areg**'. As with *jump*, a label identifier may be used as argument to this instruction.
eqc	Test if **Areg** equals constant; (result 'true' or 'false', placed in **Areg**).
j	Jump: the argument may be an identifier indicating a label for the jump to go to; the assembler will compute the displacement required.
ldc	Load constant.
ldl	Load local word
ldlp	Load pointer to local word
ldnl	Load non-local word
ldnlp	Load pointer to non-local word
opr	'operate': the argument to this instruction is a code indicating a zero-operand *indirect* instruction to be executed. Most of the transputer instruction set is made up of these indirect instructions. Normally you would use the mnemonic for the specific indirect instruction which you require: the assembler will encode this as an *opr* instruction on your behalf. However, it is possible to use *opr* explicitly, for example to synthesize the instruction sequence for a new indirect instruction not supported by the T414 and T800 transputers.
stl	Store local word
stnl	Store non-local word

A.3 Operations

The instructions in this section are all *indirect* instructions built out of the *opr* instruction. None of these instructions take an argument; instead, they work solely with the transputer evaluation stack.

The arithmetic instructions take their operands from the top of the evaluation stack (**Areg**, **Breg**) and push the result value back on the stack in **Areg**.

add	Add
alt	Alt start
altend	Alt end
altwt	Alt wait
and	Bit-wise and
bcnt	Byte count

bsub	Byte subscript (**Areg** = **Areg** + **Breg**)
ccnt1	Check count from 1
clrhalterr	Clear halt-on-error
csngl	Check single
csub0	Check subscript from 0
cword	Check word
diff	Difference
disc	Disable channel
diss	Disable skip
dist	Disable timer
div	Divide
enbc	Enable channel
enbs	Enable skip
enbt	Enable timer
endp	End process
fmul	Fractional multiply (32-bit processors only)
gajw	General adjust workspace
gcall	General call (swap **Areg**↔**Iptr**)
gt	Greater than (result 'true' or 'false', placed in **Areg**)
in	Input message
ladd	Long add
lb	Load byte
ldiff	Long difference
ldiv	Long divide
ldpi	Load pointer to instruction (**Areg** is byte displacement from **Iptr**)
ldpri	Load current priority
ldtimer	Load timer
lend	Loop end
lmul	Long multiply
lshl	Long shift left
lshr	Long shift right
lsub	Long subtract
lsum	Long sum
mint	Minimum integer
move	Move block of memory (src: **Creg** dest: **Breg** len: **Areg**)
mul	Multiply

norm	Normalize
not	Bit-wise not
or	Bit-wise inclusive or
out	Output message
outbyte	Output byte
outword	Output word
prod	Product
rem	Remainder
resetch	Reset channel
ret	Return
rev	Reverse top two stack elements
runp	Run process
saveh	Save high priority queue registers
savel	Save low priority queue registers
sb	Store byte
seterr	Set error
sethalterr	Set halt-on-error
shl	Shift left
shr	Shift right
startp	Start process
sthb	Store high priority back pointer
sthf	Store high priority front pointer
stlb	Store low priority back pointer
stlf	Store high priority back pointer
stoperr	Stop on error
stopp	Stop process
sttimer	Store timer
sub	Subtract
sum	Sum
talt	Timer alt start
taltwt	Timer alt wait
testerr	Test error false and clear
testhalterr	Test halt-on-error
testpranal	Test processor analyzing
tin	Timer input
wcnt	Word count

wsub	Word subscript (Areg = Areg + 4*Breg)
xdble	Extend to double
xor	Bit-wise exclusive or
xword	Extend to word

A.4 Additional instructions for T400, T414, T425 and TB

The indirect instructions in this section may only be executed on a T400, T414 or T425 processor, although the compiler will accept them in assembly code even when compiling for a different processor.

cflerr	Check single-length floating-point infinity or not-a-number
ldinf	Load single-length infinity
postnormsn	Post-normalize correction of single-length floating-point number
roundsn	Round single-length floating-point number
unpacksn	Unpack single-length floating-point number

A.5 Additional instructions for IMS T800, T801 and T805

The instructions in this section may only be executed on T800, T801 and T805 processors, although the compiler will accept them in assembly code even when compiling for a different processor.

A.5.1 Floating–point instructions

The indirect instructions in this section provide access to the T8 series built-in floating-point processor. Note that the instructions beginning with '*fpu* . . .' are doubly indirect: they are accessed by loading an *entry code* constant with a *ldc* instruction, then executing an *fpentry* instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the *fpentry* instruction is also available.

The floating-point load and store instructions use the *integer* Areg as a pointer to the operand location.

fpadd	Floating-point add
fpb32tor64	Convert 32-bit unsigned integer to 64-bit real
fpchkerr	Check floating error
fpdiv	Floating-point divide
fpdup	Floating duplicate
fpentry	Floating-point unit entry: used to synthesize the '*fpu* . . .' instructions.

fpeq	Floating-point equality
fpgt	Floating-point greater than
fpi32tor32	Convert 32-bit integer to 32-bit real
fpi32tor64	Convert 32-bit integer to 64-bit real
fpint	Round to floating integer
fpldnladddb	Floating load non-local and add double
fpldnladdsn	Floating load non-local and add single
fpldnldb	Floating load non-local double
fpldnldbi	Floating load non-local indexed double
fpldnlmuldb	Floating load non-local and multiply double
fpldnlmulsn	Floating load non-local and multiply single
fpldnlsn	Floating load non-local single
fpldnlsni	Floating load non-local indexed single
fpldzerodb	Floating load zero double
fpldzerosn	Load zero single
fpmul	Floating-point multiply
fpnan	Floating-point not-a-number
fpnotfinite	Floating-point finite
fpordered	Floating-point orderability
fpremfirst	Floating-point remainder first step
fpremstep	Floating-point remainder iteration step
fprev	Floating reverse
fprtoi32	Convert floating to 32-bit integer
fpstnldb	Floating store non-local double
fpstnli32	Store non-local int32
fpstnlsn	Floating store non-local single
fpsub	Floating-point subtract
fptesterr	Test floating error false and clear
fpuabs	Floating-point absolute
fpuchki32	Check in range of 32-bit integer
fpuchki64	Check in range of 64-bit integer
fpuclrerr	Clear floating error
fpudivby2	Divide by 2.0
fpuexpdec32	Divide by 2^{32}
fpuexpinc32	Multiply by 2^{32}
fpumulby2	Multiply by 2.0

fpunoround	Convert 64-bit real to 32-bit real without rounding
fpur32tor64	Convert single to double
fpur64tor32	Convert double to single
fpurm	Set rounding mode to round minus
fpurn	Set rounding mode to round nearest
fpurp	Set rounding mode to round positive
fpurz	Set rounding mode to round zero
fpuseterr	Set floating error
fpusqrtfirst	Floating-point square root first step
fpusqrtlast	Floating-point square root end
fpusqrtstep	Floating-point square root step

A.6 Additional instructions for IMS T225, T400, T425, T800, T801, T805

The indirect instructions in this section supplement the T414's integer instruction set.

bitcnt	Count the number of bits set in a word
bitrevnbits	Reverse bottom *n* bits in a word
bitrevword	Reverse bits in a word
crcbyte	Calculate CRC on byte
crcword	Calculate Cyclic Redundancy Check (CRC) on word
dup	Duplicate top of stack
wsubdb	Form double-word subscript

The following 2-dimensional block move instructions apply to the **IMS T400, T425, T800, T801 and T805** only:

move2dall	2-dimensional block copy
move2dinit	Initialize data for 2-dimensional block move
move2dnonzero	2-dimensional block copy non-zero bytes
move2dzero	2-dimensional block copy zero bytes

A.7 Additional instructions for the IMS T225, T400, T425, T801 and T805

The indirect instructions listed in this section provide debugging and general support functions.

clrj0break	Clear jump 0 break enable flag
setj0break	Set jump 0 break enable flag

testj0break	Test if jump 0 break flag is set
timerdisableh	Disable high priority timer interrupt
timerdisablel	Disable low priority timer interrupt
timerenableh	Enable high priority timer interrupt
timerenablel	Enable low priority timer interrupt
ldmemstartval	Load value of **MemStart** address
pop	Pop processor stack
lddevid	Load device identity

B Configuration language syntax

This appendix defines the syntax of the configuration language. A full description of the language and how to use it can be found in Chapter 6.

B.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly, the form is as follows:

- Terminal strings of the language — those not built up by rules of the language — are printed in teletype font e.g. `node`.
- Each phrase definition consists of an equality expression built up using a double colon and an equals sign to separate the two sides e.g. '::='.
- Alternatives are separated by vertical bars ('/').
- Optional sequences are enclosed in square brackets ('[' and ']').
- Items which may be repeated zero or more times appear in braces ('{' and '}').
- $\{_0, x\}$ represents a list of zero or more items of type 'x' separated by commas.
- $\{_1, x\}$ represents a list of one or more items of type 'x' separated by commas.

B.2 Implementation details

- Subscript ranges for arrays are dependent on the word length of the machine running the configurer. For 16-bit machines the range is 0 to 2^{15}–1, for 32-bit machines the range is 0 to 2^{31}–1.
- Each line in the source configuration file should not exceed 1024 characters, not including leading and following white space.
- The maximum number of dimensions for an identifier or array constant is 64.

B.3 Reserved words

This section defines the set of reserved words, and predefined types, attributes and constants, that are defined in the configuration language.

B.3.1 Keywords

Reserved words cannot be used by the programer as identifiers in the configuration description.

The reserved words are as follows:

`by`	`char`	`connect`	`connection`
`define`	`double`	`edge`	`else`
`float`	`for`	`if`	`input`
`int`	`node`	`on`	`output`
`place`	`rep`	`size`	`to`
`use`	`val`		

B.3.2 Pre–defined attributes

Node attributes

The `element` attribute used for defining the type of a `node` can take the following values:

- `processor` - the node is a *processor* in a *hardware* network.
- `process` - the node is a *process* in a *software* network.

Note: The names of node attributes are not reserved words and can be freely used as general purpose identifiers by the programmer.

Processor attributes

The attributes defined for nodes of type `processor` are as follows:

- `link` - used by processor nodes to define interconnection. Only defined if the `type` attribute has already been defined.
- `memory` - used by processor nodes to define memory size.
- `reserved` - used by processor nodes to reserve memory.
- `router` - used by processor nodes to control the virtual routing decisions of the configurer. The `router` attribute can take the following sub–attributes: `linkquota`, `routecost` and `tolerance`.
    - `linkquota` - suggests the maximum number of links on the associated processor that should be used by the virtual channel routing system.
    - `routecost` - defines within the range `MIN_COST` to `MAX_COST` inclusive the associated cost of through-routing data through this processor for other processor's virtual channel traffic.
    - `tolerance` - controls with any value between `ZERO_TOLERANCE` and `MAX_TOLERANCE` inclusive how much the particular processor

concerned can be used for the provision of load-sharing through-routing paths for other processors.

- **type** - used by processor nodes to define processor type. Processor types predefined in standard include files are as follows:

```
T400    T414    T425    T426
T800    T801    T805
T212    T222    T225
M212
```

Process attributes

The attribute names currently defined for nodes of type **process** are:

- **heapsize** - used by the process nodes to specify the size of the heap data segment used by the process.
- **interface** - used by process nodes to define the type and the default values of parameters to be passed into the process when the process starts executing.
- **location** - used by process nodes to specify the absolute locations of their code and data segments. The **location** attribute can take the following sub-attributes:

```
code  stack  static  heap  vector
```

- **nodebug** - used by process nodes to notify the *advanced toolset* debugger not to debug the process.
- **noprofile** - used by process nodes to notify the *advanced toolset* profiler not to profile the process.
- **order** - used by process nodes to specify the ordering of their code and data segments. The **order** attribute can take the following sub-attributes:

```
code  stack  static  heap  vector
```

- **priority** - used by process nodes to specify the priority of the process.
- **stacksize** - used by the process nodes to specify the size of the stack data segment used by the process.

B.4 Predefinitions

The following definitions are read by the configurer from the standard include file `setconf.inc` at invocation.

B.4.1 Constants

```
val FALSE 0;
val TRUE 1;

val false 0;
val true 1;

val HIGH 0;
val LOW 1;

val high 0;
val low 1;


val MIN_COST          1;
val DEFAULT_COST      1000;
val MAX_COST          1000000;
val INFINITE_COST     1000001;

val ZERO_TOLERANCE   0;
val DEFAULT_TOLERANCE  1;
val MAX_TOLERANCE     1000000;

val ROUTER_ORDER      -20000;
val MUXER_ORDER       -10000;

val min_cost          1;
val default_cost      1000;
val max_cost          1000000;
val infinite_cost     1000001;

val zero_tolerance   0;
val default_tolerance  1;
val max_tolerance     1000000;

val router_order      -20000;
val muxer_order       -10000;
```

`TRUE`, `true`, `FALSE`, and `false` are used in expressions where a boolean value is needed.

`HIGH`, `high`, `LOW`, and `low` can be used to define the execution priority for a process.

The remaining constants are used to influence the performance of the virtual routing network. See chapter 9.

B.4.2 Types

```
define node (element = "processor") processor;

define node (element = "processor", type = "T805") t805;
define node (element = "processor", type = "T801") t801;
define node (element = "processor", type = "T800") t800;
define node (element = "processor", type = "T426") t426;
define node (element = "processor", type = "T425") t425;
define node (element = "processor", type = "T414") t414;
define node (element = "processor", type = "T400") t400;
define node (element = "processor", type = "T225") t225;
define node (element = "processor", type = "T222") t222;
define node (element = "processor", type = "T212") t212;
define node (element = "processor", type = "M212") m212;

define node (element = "processor") PROCESSOR;

define node (element = "processor", type = "T805") T805;
define node (element = "processor", type = "T801") T801;
define node (element = "processor", type = "T800") T800;
define node (element = "processor", type = "T426") T426;
define node (element = "processor", type = "T425") T425;
define node (element = "processor", type = "T414") T414;
define node (element = "processor", type = "T400") T400;
define node (element = "processor", type = "T225") T225;
define node (element = "processor", type = "T222") T222;
define node (element = "processor", type = "T212") T212;
define node (element = "processor", type = "M212") M212;

define node (element = "process") process;
define node (element = "process", priority = high) highprocess;
define node (element = "process", priority = low) lowprocess;

define node (element = "process") PROCESS;
define node (element = "process", priority = HIGH) HIGHPROCESS;
define node (element = "process", priority = LOW) LOWPROCESS;
```

B.4.3 Declarations

One declaration is defined within `setconf.inc`:

```
edge host;
```

B.5 Language syntax

B.5.1 Configuration

```
configuration      ::=  config-item {config-item}

config-item        ::=  declaration
                   |    replicator
                   |    conditional
                   |    directive

declaration        ::=  node-decl
                   |    node-attr-decl
                   |    nodedef-decl
                   |    connect-decl
                   |    edge-decl
                   |    connector-decl
                   |    mapping-decl
                   |    numeric-value-decl
                   |    compound-decl
                   |    use-decl

compound-decl      ::=  { config-item {config-item} }
```

B.5.2 Language features

```
letter             ::=  A | B | ... | Z | a | b | ... | z

digit              ::=  0 | 1 | 2 | ... | 9

id-char            ::=  letter | digit | _

identifier         ::=  letter {id-char}  | _ {id-char}

comment            ::=  /* any characters except */ sequence */

directive          ::=  # file-include

file-include       ::=  include string
```

B.5.3 Expressions

```
octal-digit        ::=  0 | 1 | 2 | ... | 7

hex-digit          ::=  digit | A | B | ... | F | a | b | ... | f

octal              ::=  0 octal-digit {octal-digit}

decimal            ::=  digit {digit}
```

```
hex               ::=  0x hex-digit {hex-digit} | 0X hex-digit {hex-digit}

character-const   ::=  ' char '

char              ::=  any character except end of line and quote mark
                   |   escape-sequence

escape-sequence   ::=  \' | \" | \\ | \?
                   |   \a | \b | \f | \n | \r | \t | \v
                   |   \ octal-digit [octal-digit] [octal-digit]
                   |   \x {hex-digit}

string            ::=  " {string-char} "

string-char       ::=  any character except end of line and double quote
                       mark
                   |   escape-sequence

scale-size        ::=  k | K | l | L

int-const         ::=  decimal[scale-size] | octal | hex

sign              ::=  + | -

exponent          ::=  E [sign] decimal | e [sign] decimal

real-size         ::=  f | F | l | L

real-const        ::=  decimal . [decimal] [exponent] [real-size]
                   |   decimal exponent [real-size]
                   |   . decimal [exponent] [real-size]

array-const       ::=  { {1, exp } } | string

subscript         ::=  [ exp ] { [ exp ] }

const             ::=  int-const
                   |   real-const
                   |   character-const
                   |   array-const [subscript]

numeric type      ::=  int | float | double | char

monadic-op        ::=  + | - | ! | ~
                   |   ( numeric-type )

dyadic-op         ::=  + | - | * | / | %
                   |   & | | | ^ | << | >>
                   |   && | ||
                   |   < | > | <= | >= | == | !=

element           ::=  identifier { [subscript] . identifier } [subscript]
```

```
function-call        ::=  size ( element )

exp                  ::=  const
                     |    element
                     |    monadic-op exp
                     |    exp dyadic-op exp
                     |    exp ? exp : exp
                     |    ( exp )
                     |    function-call
```

B.5.4 Replication and conditionals

```
replicator           ::=  rep identifier = exp to exp  config–item
                     |    rep identifier = exp for exp  config–item

conditional          ::=  if exp config–item [else config–item]
```

B.5.5 Numeric value declarations

```
numeric-value-decl   ::=  val identifier exp ;
```

B.5.6 Network declarations

```
node-decl      ::=  node-type [( {1, attributes } )] {1, identifier [subscript] } ;

node-type            ::=  node
                     |    identifier

attributes           ::=  general-attr
                     |    node-attr
                     |    processor-attr
                     |    process-attr

general-attr         ::=  identifier =  exp
                     |    identifier  (  {1, general-attr }  )
                     |    identifier  (  {1, formal-attr }  )

node-attr            ::=  element = element-type

element-type         ::=  "processor"
                     |    "process"

processor-attr       ::=  type  =  processor-type
                     |    memory = exp
                     |    reserved  = exp
                     |    router  (  {1, router-attr }  )

router-attr          ::=  linkquota = exp
                     |    routecost = exp
                     |    tolerance = exp
```

```
processor-type      ::=  "M212"
                    |    "T212"
                    |    "T222"
                    |    "T225"
                    |    "T400"
                    |    "T414"
                    |    "T425"
                    |    "T426"
                    |    "T800"
                    |    "T801"
                    |    "T805"

process-attr        ::=  stacksize = exp
                    |    heapsize = exp
                    |    priority = exp
                    |    nodebug = exp
                    |    noprofile = exp
                    |    interface ( {1, formal-att } )
                    |    order ( {1, segment-att } )
                    |    location ( {1, segment-att } )

segment-attr        ::=  code = exp
                    |    heap = exp
                    |    stack = exp
                    |    static = exp
                    |    vector = exp

formal-attr         ::=  formal-type {1, identifier [subscript]  [= exp] }

formal-type         ::=  numeric-type
                    |    input
                    |    output

node-attr-decl      ::=  element ( {1, attributes } ) ;

nodedef-decl        ::=  define node-type  [( {1, attributes } )] identifier ;

connector-decl      ::=  connection   {1, identifier [subscript] } ;

connect-decl      ::=  connect element , element  [by identifier [subscript]] ;
                  |    connect element to element  [by identifier [subscript]] ;

edge-decl           ::=  edge {1, identifier [subscript] } ;
                    |    input {1, identifier [subscript] }
                    |    output {1, identifier [subscript] } ;

use-dec             ::=  use string for element ;
```

B.5.7 Mapping declarations

```
mapping-decl        ::=  place element on element ;
```

C Glossary

Alias

If two or more expressions denote the same memory address, then the expressions are aliases or one another.

Alias check

A program compilation check that ensures that names are unique within a given scope.

Analyse

A transputer input pin which forces the processor to halt at the next descheduling point, to allow the state of the processor to be read. To assert the **Analyse** input on a transputer. In the context of 'analyzing a network', to analyze all transputers in that transputer network.

Backtrace

Within the debugger an simulator tools, to move from a position within a procedure or function body to the call of that procedure or function.

Big endian

The opposite of **little endian** — see below.

Bootable code

Self-starting program code that can be loaded onto a transputer or transputer network down a **user link** and run. Bootable code is produced by `icollect` from linked units (single transputer programs) or configuration binary files (for configured programs).

Bootstrap

A transputer program, loaded from ROM or over a link after the transputer has been reset or analyzed, which initializes the processor and loads a program for execution (which may be another loader).

Capability

A text string which identifies a transputer **resource** (or resources).

Compiler library

A group of occam library routines that are used by the compiler to implement extended arithmetic and transputer system operations.

Configuration

The association of components of a program with a set of physical resources. Used in this manual to refer to the specific case of allocating software processes to processors in a network, and channels to links between processors. The term is also used, depending on the context, to describe the act of deciding on these allocations for a program, the configuration code which describes such a set of allocations, and the act of applying the configurer to a configuration description.

Configurer

The tool which assigns processes and channels on a specified configuration of transputers. The output from the tool is a configuration binary file for input to `icollect`.

Connection manager

The functionality provided by the **Linkops** part of the host file server. Provides and maintains connections to transputer systems across a network and is used by the **session manager** to select a transputer system and maintain unique access to that system.

Core dump

A memory dump. Core dumps are required as part of the process of debugging multitransputer programs that incorporate the root transputer.

Communicating Sequential Processes (CSP)

A theory and notation, developed by Professor Tony Hoare, for describing systems made up of concurrent processes which communicate via channels. The occam model of concurrency is based on CSP.

Deadlock

A state in which one or more concurrent processes can no longer proceed because of a communication interdependency.

Error modes

The compilation mode of a program that determines what happens when a program error (such as an array bounds violation) occurs. Programs are compiled by `icc` in UNIVERSAL mode, which is the mode that can be mixed with HALT and STOP code generated by other INMOS compilers.

Error signal (or error flag)

In the transputer, an external signal used to indicate that an error has occurred in a running program. Also refers to one of the system control functions on transputer networks. Error signals can be OR-ed together on transputer boards to indicate that an error has occurred in one of the transputers in a network.

Ethernet

A LAN technology based on a passive coaxial cable. It is a 10 Mbps 'best-effort' delivery system.

Extended data types

The occam data types `INT16`, `INT32`, `INT64`, `REAL32` and `REAL64`.

External memory interface (EMI)

The signals which connect a transputer to external memory, consisting of address and data buses and a number of control signals. Most of the 32 bit transputers (T4xx and T8xx) have a programmable EMI which can be configured for different types and speeds of external memory device.

Event

An input signal to the transputer which can be used an external interrupt. The event input can be treated by a process as a (zero length) communication.

Free variables

Variables which are referred to in a function or procedure, but declared outside of it.

Gateway

A dedicated computer that connects two or more networks, and routes messages between them.

Hard channels

Channels which are mapped onto **links** between processors in a transputer network (see also **soft channels**).

Host

The computer to which a transputer system is connected and which possibly also provides file system access and terminal I/O.

Host file server

A server which provides access to the filing system and terminal I/O of a host operating system.

Include file

A file containing source code which is incorporated into a program using the C `#include` (`#INCLUDE` for occam) directive. Include files are, by convention, given the `.h` extension in C; occam include files are given the extension `.inc`.

LAN (Local Area Network)

Any computer network that works over short distances at high speeds.

Library

A collection of separately compiled procedures or functions, created by the toolset librarian `ilibr`, which may be shared between parts of a program or between different programs.

Library build file

A file containing a list of input files for the librarian tool `ilibr`. Each file forms a separately loadable module in the library. Library build files should have the `.lbb` extension.

Library usage file

A file listing the libraries and separately compiled units used by another library. Library usage files must have the `.liu` extension.

Link

In the context of transputer hardware, the serial communication link between processors.

In the context of program compilation, collecting together all the compiled code for a program, resolving all references and placing the collected code into a single file.

Linker

The program or tool which links a program or compilation unit.

Linkops

The recommended INMOS link interface, used by `iserver` 1.5.

Little endian

The transputer is totally 'little endian', i.e. less significant data is always held in lower addresses. This applies to bits in bytes, bytes in words and words in memory.

Loader

Depending on the context, refers to the part of the host file server which loads a transputer network or to a small program which is loaded into a transputer, and which then distributes code to other transputers and loads a larger program on top of itself.

Makefile

An input file for a 'make' program. A makefile contains details of file dependencies and directions for rebuilding the object code. Makefiles are created for the toolset using `imakef`.

Network

Depending on context may refer to a conventional computer network or a set of interconnected transputers.

Object code

Intermediate code between source and **bootable code**. Object code cannot be directly loaded onto a transputer and run. The compiler and **linker** tools generate object code.

Peek and poke

To read (peek) and write (poke) locations in a transputer's memory via a link, while the transputer is waiting to be booted.

PostScript

PostScript is a device-independent, interpreted language for describing the layout of text and graphics on a page. It is used by a large number of printers and software applications as the standard means of transferring graphics data.

Preamble

The part of a transputer loader program that initializes the state of the processor.

Priority

In the transputer, the priority level at which the currently executing process is being run. INMOS transputers support two levels of priority, known as 'high' and 'low'.

Process

Self-contained, independently executable code.

Protocol

The pattern (type, etc.) of communications between two processes, often including communications on more than one channel. Protocols can be defined in occam and must be specified when a channel is declared.

Reset

The transputer system initialization control signal.

Root transputer (or root processor)

The processor in a transputer network which is physically connected to the host computer, and through which the transputer network is loaded.

Separate compilation

A self-contained part of a program may be separately compiled, so that only those parts of a program which have changed since the last compilation need to be re-compiled (see also **makefile**).

Server

A program running on a host computer which provides access to the filing system and terminal I/O of the host for the transputers, or access to the transputer system from the host. The server can also be used to load the program onto the network.

Session manager

That part of the **server** which maintains unique access (a session) to a transputer system when requested by a user.

Soft channels

Channels declared and used within a process running on a single transputer (see also **hard channels**). Soft channels are implemented by a single word in memory.

Standard error

The **host** system error handler. Errors directed to standard error are displayed in a host-defined way, for example, on the terminal screen. For details of how to modify standard error on the system, consult the operating system documentation.

Standard input

The **host** system input handler. Specifies the standard input device, for example the terminal keyboard or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Standard output

The **host** system output handler. Specifies the standard output device, for example, the terminal screen or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Subsystem

In transputer board architecture, the combination of the **Reset**, **Analyse** and **Error** signals which allows one board to control another board connected to its **subsystem** port.

Target transputer

The transputer on which the code is intended to run. The transputer type, or a restricted set of types defined in a transputer class, is defined when the program is compiled, using command line options.

Transputer Module (TRAM)

A range of small printed circuit boards which typically hold a transputer, some memory and, optionally, some other application specific hardware. TRAMs can be interconnected via links to build systems based on a number of motherboard architectures. For more information see the *iq* systems databook.

Usage check

A compilation check that ensures no variables are shared between parallel processes, and that enforces rules about the use of channels as unidirectional, point-to-point connections.

User link

The connection of a transputer **resource** to a **host** computer.

Vector space

The data space required for the storage of arrays within occam code (see also **workspace**).

Worm

A program that distributes itself through a network of transputers (perhaps with an unknown topology) and allows all the processors in the network to be loaded, tested or analyzed.

Workspace

The data space required by an occam process. When used in contrast to **vector space**, refers to the data space required for *scalars* within the occam code.

D Bibliography

D.1 Transputers

The transputer databook (Third Edition 1992)

INMOS Ltd, July 1992
INMOS document number 72 TRN 203 02

The military and space transputer databook (First Edition 1990)

INMOS Ltd, July 1990
INMOS document number 72 TRN 224 00

Transputer instruction set: A compiler writer's guide

INMOS Ltd
Prentice Hall 1988

Transputer Hardware and systems design

JC Hinton and AL Pinder
Prentice Hall 1993

The transputer handbook

Ian Graham and Tim King
Prentice Hall 1990

D.2 C programming

The C programming language (First Edition)

Brian W Kernighan & Dennis M Ritchie
Prentice Hall 1978

The C programming language (Second Edition — ANSI C)

Brian W Kernighan and Dennis M Ritchie
Prentice Hall 1988

C: A reference manual (Second Edition — ANSI C)

Samuel P Harbison and Guy L Steele
Prentice Hall 1987

American National Standard for Information Systems – Programming Language C

American National Standards Institute 1990
Ref. Doc. X3J11/88–159

D.3 occam programming

occam 2 reference manual

INMOS Ltd
Prentice Hall 1988

A tutorial introduction to occam programming

D Pountain and D May
Blackwell Scientific 1987

An introduction to occam 2 programming

KC Bowler, RD Kenway, GS Pawley and D Roweth
Chartwell–Bratt 1987

Programming in occam 2

A Burns
Addison–Wesley 1988

occam 2

A Gallently
Piman 1989

Programming in occam *2*

G Jones and M Goldsmith
Prentice Hall 1988

Concurrent programming in occam 2

J Wexler
Ellis Horwood 1989

D.4 INMOS technical notes

The transputer applications notebook:
Architecture and software (First Edition 1989)

INMOS Ltd, May 1989
INMOS document number 72 TRN 206 00

The transputer applications notebook:
Systems and performance (First Edition 1989)

INMOS Ltd, June 1989
INMOS document number 72 TRN 205 00

IMS B004 IBM PC add-in board

Technical note 11
INMOS document number 72 TCH 011

Notes on graphics support and performance improvements on the IMS T800

Technical note 26
INMOS document number 72 TCH 026

Security aspects of occam *2*

Technical note 33
INMOS document number 72 TCH 033

Simple real-time programming with the transputer

Technical note 51
INMOS document number 72 TCH 051

Using the occam *toolsets with non-*occam *applications*

Technical note 55
INMOS document number 72 TCH 055

D.5 Development systems

The transputer development and **iq** systems *databook* (Second Edition 1991)

INMOS Ltd, 1991
INMOS Document Number 72 TRN 219 01

IMS B300 TCPlink hardware manual

INMOS Limited, June 1991
INMOS Document Number 72 TRN 229 01

ＡＮＳＩ Ｃ ツールセット・ユーザー・マニュアル（日本語版）
（Ｄ４２１４Ｂ、Ｄ５２１４Ｂ、Ｄ７２１４Ｃバージョン）

翻訳：　新日本製鉄（株）エレクトロニクス研究所電子システム研究部
　　　　貝塚洋、田内宏明、斉藤知人、金子博昭、青柳雄大
　　　　ＳＧＳトムソン・マイクロエレクトロニクス（株）
　　　　梅本剛
発行所：　日刊工業新聞社（株）

D.6 References

Software manual for the elementary functions

WJ Cody and WM Waite
Prentice Hall 1980

The art of computer programming
2nd edition, Volume 2: *Seminumerical algorithms*

DE Knuth
Addison Wesley 1981

IEEE Standard for binary floating-point arithmetic

ANSI–IEEE Std 754–1985

Communicating sequential processes

CAR Hoare
Prentice Hall

Index